python-zstandard

Release 0.19.0

Oct 30, 2022

Contents

CHAPTER 1

Project Information

1.1 State of Project

The project is officially in beta state. The author is reasonably satisfied that functionality works as advertised. **There will be some backwards incompatible changes before 1.0, probably in the 0.9 release.** This may involve renaming the main module from *zstd* to *zstandard* and renaming various types and methods. Pin the package version to prevent unwanted breakage when this change occurs!

This project is vendored and distributed with Mercurial 4.1, where it is used in a production capacity.

There is continuous integration for Python versions and 3.6+ on Linux x86_x64 and Windows x86 and x86_64. The author is reasonably confident the extension is stable and works as advertised on these platforms.

The CFFI bindings are mostly feature complete. Where a feature is implemented in CFFI, unit tests run against both C extension and CFFI implementation to ensure behavior parity.

1.2 Comparison to Other Python Bindings

https://pypi.python.org/pypi/zstd is an alternate Python binding to Zstandard. At the time this was written, the latest release of that package (1.4.8) only exposed the simple APIs for compression and decompression. This package exposes much more of the zstd API, including streaming and dictionary compression. This package also has CFFI support.

https://github.com/animalize/pyzstd is an alternate Python binding to Zstandard. At the time this was written, the latest release of that package (0.14.1) exposed a fraction of the functionality in this package. There may be some minor features in `pyzstd` not found in this package. But those features could be added easily if someone made a feature request. Also, `pyzstd` lacks CFFI support, so it won't run on PyPy.

1.3 Performance

zstandard is a highly tunable compression algorithm. In its default settings (compression level 3), it will be faster at compression and decompression and will have better compression ratios than zlib on most data sets. When tuned for speed, it approaches lz4's speed and ratios. When tuned for compression ratio, it approaches lzma ratios and compression speed, but decompression speed is much faster. See the official zstandard documentation for more.

zstandard and this library support multi-threaded compression. There is a mechanism to compress large inputs using multiple threads.

The performance of this library is usually very similar to what the zstandard C API can deliver. Overhead in this library is due to general Python overhead and can't easily be avoided by *any* zstandard Python binding. This library exposes multiple APIs for performing compression and decompression so callers can pick an API suitable for their need. Contrast with the compression modules in Python's standard library (like `zlib`), which only offer limited mechanisms for performing operations. The API flexibility means consumers can choose to use APIs that facilitate zero copying or minimize Python object creation and garbage collection overhead.

This library is capable of single-threaded throughputs well over 1 GB/s. For exact numbers, measure yourself. The source code repository has a `bench.py` script that can be used to measure things.

1.4 Bundling of Zstandard Source Code

The source repository for this project contains a vendored copy of the Zstandard source code. This is done for a few reasons.

First, Zstandard is relatively new and not yet widely available as a system package. Providing a copy of the source code enables the Python C extension to be compiled without requiring the user to obtain the Zstandard source code separately.

Second, Zstandard has both a stable *public* API and an *experimental* API. The *experimental* API is actually quite useful (contains functionality for training dictionaries for example), so it is something we wish to expose to Python. However, the *experimental* API is only available via static linking. Furthermore, the *experimental* API can change at any time. So, control over the exact version of the Zstandard library linked against is important to ensure known behavior.

1.5 Note on Zstandard's *Experimental* API

Many of the Zstandard APIs used by this module are marked as *experimental* within the Zstandard project.

It is unclear how Zstandard's C API will evolve over time, especially with regards to this *experimental* functionality. We will try to maintain backwards compatibility at the Python API level. However, we cannot guarantee this for things not under our control.

Since a copy of the Zstandard source code is distributed with this module and since we compile against it, the behavior of a specific version of this module should be constant for all of time. So if you pin the version of this module used in your projects (which is a Python best practice), you should be shielded from unwanted future changes.

1.6 Donate

A lot of time has been invested into this project by the author.

If you find this project useful and would like to thank the author for their work or commission a feature, consider donating some money. Any amount is appreciated. This can be done through GitHub Sponsors at https://github.com/sponsors/indygreg.

CHAPTER 2

Installing

This package is uploaded to PyPI at https://pypi.python.org/pypi/zstandard. So, to install this package:

```
$ pip install zstandard
```

Binary wheels are made available for some platforms. If you need to install from a source distribution, all you should need is a working C compiler and the Python development headers/libraries. On many Linux distributions, you can install a `python-dev` or `python-devel` package to provide these dependencies.

Packages are also uploaded to Anaconda Cloud at https://anaconda.org/indygreg/zstandard. See that URL for how to install this package with `conda`.

2.1 Requirements

This package is designed to run with Python 3.6, 3.7, 3.8, and 3.9 on common platforms (Linux, Windows, and OS X). On PyPy (both PyPy2 and PyPy3) we support version 6.0.0 and above. x86 and x86_64 are well-tested on Windows. Only x86_64 is well-tested on Linux and macOS.

2.2 CFFI Backend

In order to build/run the CFFI backend/bindings (as opposed to the C/Rust backend/bindings), you will need the `cffi` package installed. The `cffi` package is listed as an optional dependency in `setup.py` and may not get picked up by your packaging tools.

If you wish to use the CFFI backend (or have to use it since your Python distribution doesn't support compiled extensions using the Python C API - this is the case for PyPy for example), be sure you have the `cffi` package installed.

One way to do this is to depend on the `zstandard[cffi]` dependency. e.g. `pip install 'zstandard[cffi]'` or add `zstandard[cffi]` to your pip requirements file.

2.3 Legacy Format Support

To enable legacy zstd format support which is needed to handle files compressed with zstd < 1.0 you need to provide an installation option:

```
$ pip install zstandard --install-option="--legacy"
```

and since pip 7.0 it is possible to have the following line in your requirements.txt:

```
zstandard --install-option="--legacy"
```

2.4 All Install Arguments

`setup.py` accepts the following arguments for influencing behavior:

`--legacy` Enable legacy zstd format support in order to read files produced with zstd < 1.0.

`--system-zstd` Attempt to link against the zstd library present on the system instead of the version distributed with the extension.

The Python extension only supports linking against a specific version of zstd. So if the system version differs from what is expected, a build or runtime error will result.

`--warning-as-errors` Treat all compiler warnings as errors.

`--no-c-backend` Do not compile the C-based backend.

`--no-cffi-backend` Do not compile the CFFI-based backend.

`--rust-backend` Compile the Rust backend (not yet feature complete).

If you invoke `setup.py`, simply pass the aforementioned arguments. e.g. `python3.9 setup.py --no-cffi-backend`. If using `pip`, use the `--install-option` argument. e.g. `python3.9 -m pip install zstandard --install-option --warning-as-errors`. Or in a pip requirements file: `zstandard --install-option="--rust-backend"`.

In addition, the following environment variables are recognized:

`ZSTD_EXTRA_COMPILER_ARGS` Extra compiler arguments to compile the C backend with.

`ZSTD_WARNINGS_AS_ERRORS` Equivalent to `setup.py --warnings-as-errors`.

2.5 Building Against External libzstd

By default, this package builds and links against a single file `libzstd` bundled as part of the package distribution. This copy of `libzstd` is statically linked into the extension.

It is possible to point `setup.py` at an external (typically system provided) `libzstd`. To do this, simply pass `--system-zstd` to `setup.py`. e.g.

`python3.9 setup.py --system-zstd` or `python3.9 -m pip install zstandard --install-option="--system-zstd"`.

When building against a system libzstd, you may need to specify extra compiler arguments to help Python's build system find the external library. These can be specified via the `ZSTD_EXTRA_COMPILER_ARGS` environment variable. e.g. `ZSTD_EXTRA_COMPILER_ARGS="-I/usr/local/include" python3.9 setup.py --system-zstd`.

`python-zstandard` can be sensitive about what version of `libzstd` it links against. For best results, point this package at the exact same version of `libzstd` that it bundles. See the bundled `zstd/zstd.h` or `zstd/zstdlib.c` for which version that is.

When linking against an external `libzstd`, not all package features may be available. Notably, the `multi_compress_to_buffer()` and `multi_decompress_to_buffer()` APIs are not available, as these rely on private symbols in the `libzstd` C source code, which require building against private header files to use.

CHAPTER 3

Concepts

It is useful to have a basic understanding of how Zstandard works in order to optimally use this library. In addition, there are some low-level Python concepts that are worth explaining to aid understanding. This article aims to provide that knowledge.

3.1 Zstandard Frames and Compression Format

Compressed zstandard data almost always exists within a container called a *frame*. (For the technically curious, see the specification.)

The frame contains a header and optional trailer. The header contains a magic number to self-identify as a zstd frame and a description of the compressed data that follows.

Among other things, the frame *optionally* contains the size of the decompressed data the frame represents, a 32-bit checksum of the decompressed data (to facilitate verification during decompression), and the ID of the dictionary used to compress the data.

Storing the original content size in the frame (`write_content_size=True` to `ZstdCompressor`) is important for performance in some scenarios. Having the decompressed size stored there (or storing it elsewhere) allows decompression to perform a single memory allocation that is exactly sized to the output. This is faster than continuously growing a memory buffer to hold output.

3.2 Compression and Decompression Contexts

In order to perform a compression or decompression operation with the zstd C API, you need what's called a *context*. A context essentially holds configuration and state for a compression or decompression operation. For example, a compression context holds the configured compression level.

Contexts can be reused for multiple operations. Since creating and destroying contexts is not free, there are performance advantages to reusing contexts.

The `ZstdCompressor` and `ZstdDecompressor` types are essentially wrappers around these contexts in the zstd C API.

3.3 One-shot And Streaming Operations

A compression or decompression operation can either be performed as a single *one-shot* operation or as a continuous *streaming* operation.

In one-shot mode (the *simple* APIs provided by the Python interface), **all** input is handed to the compressor or decompressor as a single buffer and **all** output is returned as a single buffer.

In streaming mode, input is delivered to the compressor or decompressor as a series of chunks via multiple function calls. Likewise, output is obtained in chunks as well.

Streaming operations require an additional *stream* object to be created to track the operation. These are logical extensions of *context* instances.

There are advantages and disadvantages to each mode of operation. There are scenarios where certain modes can't be used. See the `Choosing an API` section for more.

3.4 Dictionaries

A compression *dictionary* is essentially data used to seed the compressor state so it can achieve better compression. The idea is that if you are compressing a lot of similar pieces of data (e.g. JSON documents or anything sharing similar structure), then you can find common patterns across multiple objects then leverage those common patterns during compression and decompression operations to achieve better compression ratios.

Dictionary compression is generally only useful for small inputs - data no larger than a few kilobytes. The upper bound on this range is highly dependent on the input data and the dictionary.

3.5 Python Buffer Protocol

Many functions in the library operate on objects that implement Python's buffer protocol.

The *buffer protocol* is an internal implementation detail of a Python type that allows instances of that type (objects) to be exposed as a raw pointer (or buffer) in the C API. In other words, it allows objects to be exposed as an array of bytes.

From the perspective of the C API, objects implementing the *buffer protocol* all look the same: they are just a pointer to a memory address of a defined length. This allows the C API to be largely type agnostic when accessing their data. This allows custom types to be passed in without first converting them to a specific type.

Many Python types implement the buffer protocol. These include `bytes`, `bytearray`, `array.array`, `io.BytesIO`, `mmap.mmap`, and `memoryview`.

3.6 Requiring Output Sizes for Non-Streaming Decompression APIs

Non-streaming decompression APIs require that either the output size is explicitly defined (either in the zstd frame header or passed into the function) or that a max output size is specified. This restriction is for your safety.

The *one-shot* decompression APIs store the decompressed result in a single buffer. This means that a buffer needs to be pre-allocated to hold the result. If the decompressed size is not known, then there is no universal good default size

to use. Any default will fail or will be highly sub-optimal in some scenarios (it will either be too small or will put stress on the memory allocator to allocate a too large block).

A *helpful* API may retry decompression with buffers of increasing size. While useful, there are obvious performance disadvantages, namely redoing decompression N times until it works. In addition, there is a security concern. Say the input came from highly compressible data, like 1 GB of the same byte value. The output size could be several magnitudes larger than the input size. An input of <100KB could decompress to >1GB. Without a bounds restriction on the decompressed size, certain inputs could exhaust all system memory. That's not good and is why the maximum output size is limited.

CHAPTER 4

API Usage

To interface with Zstandard, simply import the `zstandard` module:

```
import zstandard
```

It is a popular convention to alias the module as a different name for brevity:

```
import zstandard as zstd
```

This module attempts to import and use either the C extension or CFFI implementation. On Python platforms known to support C extensions (like CPython), it raises an ImportError if the C extension cannot be imported. On Python platforms known to not support C extensions (like PyPy), it only attempts to import the CFFI implementation and raises ImportError if that can't be done. On other platforms, it first tries to import the C extension then falls back to CFFI if that fails and raises ImportError if CFFI fails.

To change the module import behavior, a `PYTHON_ZSTANDARD_IMPORT_POLICY` environment variable can be set. The following values are accepted:

`default` The behavior described above.

`cffi_fallback` Always try to import the C extension then fall back to CFFI if that fails.

`cext` Only attempt to import the C extension.

`cffi` Only attempt to import the CFFI implementation.

In addition, the `zstandard` module exports a `backend` attribute containing the string name of the backend being used. It will be one of `cext` or `cffi` (for *C extension* and *cffi*, respectively).

Note: The documentation in this section makes references to various zstd concepts and functionality. See *Concepts* for more details.

4.1 Choosing an API

There are multiple APIs for performing compression and decompression. This is because different applications have different needs and this library wants to facilitate optimal use in as many use cases as possible.

From a high-level, APIs are divided into *one-shot* and *streaming*: either you are operating on all data at once or you operate on it piecemeal.

The *one-shot* APIs are useful for small data, where the input or output size is known. (The size can come from a buffer length, file size, or stored in the zstd frame header.) A limitation of the *one-shot* APIs is that input and output must fit in memory simultaneously. For say a 4 GB input, this is often not feasible.

The *one-shot* APIs also perform all work as a single operation. So, if you feed it large input, it could take a long time for the function to return.

The streaming APIs do not have the limitations of the simple API. But the price you pay for this flexibility is that they are more complex than a single function call.

The streaming APIs put the caller in control of compression and decompression behavior by allowing them to directly control either the input or output side of the operation.

With the *streaming input*, *compressor*, and *decompressor* APIs, the caller has full control over the input to the compression or decompression stream. They can directly choose when new data is operated on.

With the *streaming ouput* APIs, the caller has full control over the output of the compression or decompression stream. It can choose when to receive new data.

When using the *streaming* APIs that operate on file-like or stream objects, it is important to consider what happens in that object when I/O is requested. There is potential for long pauses as data is read or written from the underlying stream (say from interacting with a filesystem or network). This could add considerable overhead.

4.2 Thread and Object Reuse Safety

Unless stated otherwise, `ZstdCompressor` and `ZstdDecompressor` instances cannot be used for temporally overlapping (de)compression operations. i.e. if you start a (de)compression operation on an instance or a helper object derived from it, it isn't safe to start another (de)compression operation from the same instance until the first one has finished.

`ZstdCompressor` and `ZstdDecompressor` instances have no guarantees about thread safety. Do not operate on the same `ZstdCompressor` and `ZstdDecompressor` instance simultaneously from different threads. It is fine to have different threads call into a single instance, just not at the same time.

Objects derived from `ZstdCompressor` and `ZstdDecompressor` that perform (de)compression operations (such as `ZstdCompressionReader` and `ZstdDecompressionWriter`) are bound to the `ZstdCompressor` or `ZstdDecompressor` from which they came and are therefore not thread safe by extension.

Some operations require multiple function calls to complete. e.g. streaming operations. A single `ZstdCompressor` or `ZstdDecompressor` cannot be used for simultaneously active operations. e.g. you must not start a streaming operation when another streaming operation is already active.

If you need to perform multiple compression or decompression operations in parallel, you MUST construct multiple `ZstdCompressor` or `ZstdDecompressor` instances so each independent operation has its own `ZstdCompressor` or `ZstdDecompressor` instance.

The C extension releases the GIL during non-trivial calls into the zstd C API. Non-trivial calls are notably compression and decompression. Trivial calls are things like parsing frame parameters. Where the GIL is released is considered an implementation detail and can change in any release.

APIs that accept bytes-like objects don't enforce that the underlying object is read-only. However, it is assumed that the passed object is read-only for the duration of the function call. It is possible to pass a mutable object (like a `bytearray`) to e.g. `ZstdCompressor.compress()`, have the GIL released, and mutate the object from another thread. Such a race condition is a bug in the consumer of python-zstandard. Most Python data types are immutable, so unless you are doing something fancy, you don't need to worry about this.

4.3 Performance Considerations

The `ZstdCompressor` and `ZstdDecompressor` types maintain state to a persistent compression or decompression *context*. Reusing a `ZstdCompressor` or `ZstdDecompressor` instance for multiple operations is faster than instantiating a new `ZstdCompressor` or `ZstdDecompressor` for each operation. The differences are magnified as the size of data decreases. For example, the difference between *context* reuse and non-reuse for 100,000 100 byte inputs will be significant (possibly over 10x faster to reuse contexts) whereas 10 100,000,000 byte inputs will be more similar in speed (because the time spent doing compression dwarfs time spent creating new *contexts*).

CHAPTER 5

Compression APIs

5.1 ZstdCompressor

class zstandard.**ZstdCompressor**(*level=3*, *dict_data=None*, *compression_params=None*, *write_checksum=None*, *write_content_size=None*, *write_dict_id=None*, *threads=0*)

Create an object used to perform Zstandard compression.

Each instance is essentially a wrapper around a `ZSTD_CCtx` from zstd's C API.

An instance can compress data various ways. Instances can be used multiple times. Each compression operation will use the compression parameters defined at construction time.

`compression_params` is mutually exclusive with `level`, `write_checksum`, `write_content_size`, `write_dict_id`, and `threads`.

Assume that each `ZstdCompressor` instance can only handle a single logical compression operation at the same time. i.e. if you call a method like `stream_reader()` to obtain multiple objects derived from the same `ZstdCompressor` instance and attempt to use them simultaneously, errors will likely occur.

If you need to perform multiple logical compression operations and you can't guarantee those operations are temporally non-overlapping, you need to obtain multiple `ZstdCompressor` instances.

Unless specified otherwise, assume that no two methods of `ZstdCompressor` instances can be called from multiple Python threads simultaneously. In other words, assume instances are not thread safe unless stated otherwise.

Parameters

- **level** – Integer compression level. Valid values are all negative integers through 22. Lower values generally yield faster operations with lower compression ratios. Higher values are generally slower but compress better. The default is 3, which is what the `zstd` CLI uses. Negative levels effectively engage `--fast` mode from the `zstd` CLI.
- **dict_data** –

  A `ZstdCompressionDict` to be used to compress with dictionary data.

- **compression_params** – A `ZstdCompressionParameters` instance defining low-level compression parameters. If defined, this will overwrite the `level` argument.
- **write_checksum** – If True, a 4 byte content checksum will be written with the compressed data, allowing the decompressor to perform content verification.
- **write_content_size** – If True (the default), the decompressed content size will be included in the header of the compressed data. This data will only be written if the compressor knows the size of the input data.
- **write_dict_id** – Determines whether the dictionary ID will be written into the compressed data. Defaults to True. Only adds content to the compressed data if a dictionary is being used.
- **threads** – Number of threads to use to compress data concurrently. When set, compression operations are performed on multiple threads. The default value (0) disables multi-threaded compression. A value of `-1` means to set the number of threads to the number of detected logical CPUs.

chunker(*size=-1*, *chunk_size=131591*)

Create an object for iterative compressing to same-sized chunks.

This API is similar to *ZstdCompressor.compressobj()* but has better performance properties.

Parameters

- **size** – Size in bytes of data that will be compressed.
- **chunk_size** – Size of compressed chunks.

Returns *ZstdCompressionChunker*

compress(*data*)

Compress data in a single operation.

This is the simplest mechanism to perform compression: simply pass in a value and get a compressed value back. It is almost the most prone to abuse.

The input and output values must fit in memory, so passing in very large values can result in excessive memory usage. For this reason, one of the streaming based APIs is preferred for larger values.

Parameters **data** – Source data to compress

Returns Compressed data

```
>>> cctx = zstandard.ZstdCompressor()
>>> compressed = cctx.compress(b"data to compress")
```

compressobj(*size=-1*)

Obtain a compressor exposing the Python standard library compression API.

See *ZstdCompressionObj* for the full documentation.

Parameters **size** – Size in bytes of data that will be compressed.

Returns *ZstdCompressionObj*

copy_stream(*ifh*, *ofh*, *size=-1*, *read_size=131072*, *write_size=131591*)

Copy data between 2 streams while compressing it.

Data will be read from `ifh`, compressed, and written to `ofh`. `ifh` must have a `read(size)` method. `ofh` must have a `write(data)` method.

```
>>> cctx = zstandard.ZstdCompressor()
>>> with open(input_path, "rb") as ifh, open(output_path, "wb") as ofh:
...     cctx.copy_stream(ifh, ofh)
```

It is also possible to declare the size of the source stream:

```
>>> cctx = zstandard.ZstdCompressor()
>>> cctx.copy_stream(ifh, ofh, size=len_of_input)
```

You can also specify how large the chunks that are `read()` and `write()` from and to the streams:

```
>>> cctx = zstandard.ZstdCompressor()
>>> cctx.copy_stream(ifh, ofh, read_size=32768, write_size=16384)
```

The stream copier returns a 2-tuple of bytes read and written:

```
>>> cctx = zstandard.ZstdCompressor()
>>> read_count, write_count = cctx.copy_stream(ifh, ofh)
```

Parameters

- **ifh** – Source stream to read from
- **ofh** – Destination stream to write to
- **size** – Size in bytes of the source stream. If defined, compression parameters will be tuned for this size.
- **read_size** – Chunk sizes that source stream should be `read()` from.
- **write_size** – Chunk sizes that destination stream should be `write()` to.

Returns 2-tuple of ints of bytes read and written, respectively.

frame_progression()

Return information on how much work the compressor has done.

Returns a 3-tuple of (ingested, consumed, produced).

```
>>> cctx = zstandard.ZstdCompressor()
>>> (ingested, consumed, produced) = cctx.frame_progression()
```

memory_size()

Obtain the memory usage of this compressor, in bytes.

```
>>> cctx = zstandard.ZstdCompressor()
>>> memory = cctx.memory_size()
```

multi_compress_to_buffer(*data*, *threads=-1*)

Compress multiple pieces of data as a single function call.

(Experimental. Not yet supported by CFFI backend.)

This function is optimized to perform multiple compression operations as as possible with as little overhead as possible.

Data to be compressed can be passed as a `BufferWithSegmentsCollection`, a `BufferWithSegments`, or a list containing byte like objects. Each element of the container will be compressed individually using the configured parameters on the `ZstdCompressor` instance.

The `threads` argument controls how many threads to use for compression. The default is `0` which means to use a single thread. Negative values use the number of logical CPUs in the machine.

The function returns a `BufferWithSegmentsCollection`. This type represents N discrete memory allocations, each holding 1 or more compressed frames.

Output data is written to shared memory buffers. This means that unlike regular Python objects, a reference to *any* object within the collection keeps the shared buffer and therefore memory backing it alive. This can have undesirable effects on process memory usage.

The API and behavior of this function is experimental and will likely change. Known deficiencies include:

- If asked to use multiple threads, it will always spawn that many threads, even if the input is too small to use them. It should automatically lower the thread count when the extra threads would just add overhead.
- The buffer allocation strategy is fixed. There is room to make it dynamic, perhaps even to allow one output buffer per input, facilitating a variation of the API to return a list without the adverse effects of shared memory buffers.

Parameters **data** – Source to read discrete pieces of data to compress.

Can be a `BufferWithSegmentsCollection`, a `BufferWithSegments`, or a `list[bytes]`.

Returns BufferWithSegmentsCollection holding compressed data.

read_to_iter(*reader*, *size=-1*, *read_size=131072*, *write_size=131591*)

Read uncompressed data from a reader and return an iterator

Returns an iterator of compressed data produced from reading from `reader`.

This method provides a mechanism to stream compressed data out of a source as an iterator of data chunks.

Uncompressed data will be obtained from `reader` by calling the `read(size)` method of it or by reading a slice (if `reader` conforms to the *buffer protocol*). The source data will be streamed into a compressor. As compressed data is available, it will be exposed to the iterator.

Data is read from the source in chunks of `read_size`. Compressed chunks are at most `write_size` bytes. Both values default to the zstd input and and output defaults, respectively.

If reading from the source via `read()`, `read()` will be called until it raises or returns an empty bytes (`b""`). It is perfectly valid for the source to deliver fewer bytes than were what requested by `read(size)`.

The caller is partially in control of how fast data is fed into the compressor by how it consumes the returned iterator. The compressor will not consume from the reader unless the caller consumes from the iterator.

```
>>> cctx = zstandard.ZstdCompressor()
>>> for chunk in cctx.read_to_iter(fh):
...     # Do something with emitted data.
```

`read_to_iter()` accepts a `size` argument declaring the size of the input stream:

```
>>> cctx = zstandard.ZstdCompressor()
>>> for chunk in cctx.read_to_iter(fh, size=some_int):
>>>     pass
```

You can also control the size that data is `read()` from the source and the ideal size of output chunks:

```
>>> cctx = zstandard.ZstdCompressor()
>>> for chunk in cctx.read_to_iter(fh, read_size=16384, write_size=8192):
>>>     pass
```

`read_to_iter()` does not give direct control over the sizes of chunks fed into the compressor. Instead, chunk sizes will be whatever the object being read from delivers. These will often be of a uniform size.

Parameters

- **reader** – Stream providing data to be compressed.
- **size** – Size in bytes of input data.
- **read_size** – Controls how many bytes are `read()` from the source.
- **write_size** – Controls the output size of emitted chunks.

Returns Iterator of `bytes`.

stream_reader(*source*, *size=-1*, *read_size=131072*, *closefd=True*)

Wrap a readable source with a stream that can read compressed data.

This will produce an object conforming to the `io.RawIOBase` interface which can be `read()` from to retrieve compressed data from a source.

The source object can be any object with a `read(size)` method or an object that conforms to the buffer protocol.

See *ZstdCompressionReader* for type documentation and usage examples.

Parameters

- **source** – Object to read source data from
- **size** – Size in bytes of source object.
- **read_size** – How many bytes to request when `read()`'ing from the source.
- **closefd** – Whether to close the source stream when the returned stream is closed.

Returns *ZstdCompressionReader*

stream_writer(*writer*, *size=-1*, *write_size=131591*, *write_return_read=True*, *closefd=True*)

Create a stream that will write compressed data into another stream.

The argument to `stream_writer()` must have a `write(data)` method. As compressed data is available, `write()` will be called with the compressed data as its argument. Many common Python types implement `write()`, including open file handles and `io.BytesIO`.

See *ZstdCompressionWriter* for more documentation, including usage examples.

Parameters

- **writer** – Stream to write compressed data to.
- **size** – Size in bytes of data to be compressed. If set, it will be used to influence compression parameter tuning and could result in the size being written into the header of the compressed data.
- **write_size** – How much data to `write()` to `writer` at a time.
- **write_return_read** – Whether `write()` should return the number of bytes that were consumed from the input.
- **closefd** – Whether to `close` the `writer` when this stream is closed.

Returns *ZstdCompressionWriter*

5.2 ZstdCompressionWriter

class zstandard.**ZstdCompressionWriter**(*compressor*, *writer*, *source_size*, *write_size*, *write_return_read*, *closefd=True*)

Writable compressing stream wrapper.

`ZstdCompressionWriter` is a write-only stream interface for writing compressed data to another stream.

This type conforms to the `io.RawIOBase` interface and should be usable by any type that operates against a *file-object* (`typing.BinaryIO` in Python type hinting speak). Only methods that involve writing will do useful things.

As data is written to this stream (e.g. via `write()`), that data is sent to the compressor. As compressed data becomes available from the compressor, it is sent to the underlying stream by calling its `write()` method.

Both `write()` and `flush()` return the number of bytes written to the object's `write()`. In many cases, small inputs do not accumulate enough data to cause a write and `write()` will return `0`.

Calling `close()` will mark the stream as closed and subsequent I/O operations will raise `ValueError` (per the documented behavior of `io.RawIOBase`). `close()` will also call `close()` on the underlying stream if such a method exists and the instance was constructed with `closefd=True`

Instances are obtained by calling *`ZstdCompressor.stream_writer()`*.

Typically usage is as follows:

```
>>> cctx = zstandard.ZstdCompressor(level=10)
>>> compressor = cctx.stream_writer(fh)
>>> compressor.write(b"chunk 0\n")
>>> compressor.write(b"chunk 1\n")
>>> compressor.flush()
>>> # Receiver will be able to decode ``chunk 0\nchunk 1\n`` at this point.
>>> # Receiver is also expecting more data in the zstd *frame*.
>>>
>>> compressor.write(b"chunk 2\n")
>>> compressor.flush(zstandard.FLUSH_FRAME)
>>> # Receiver will be able to decode ``chunk 0\nchunk 1\nchunk 2``.
>>> # Receiver is expecting no more data, as the zstd frame is closed.
>>> # Any future calls to ``write()`` at this point will construct a new
>>> # zstd frame.
```

Instances can be used as context managers. Exiting the context manager is the equivalent of calling `close()`, which is equivalent to calling `flush(zstandard.FLUSH_FRAME)`:

```
>>> cctx = zstandard.ZstdCompressor(level=10)
>>> with cctx.stream_writer(fh) as compressor:
...     compressor.write(b'chunk 0')
...     compressor.write(b'chunk 1')
...     ...
```

Important: If `flush(FLUSH_FRAME)` is not called, emitted data doesn't constitute a full zstd *frame* and consumers of this data may complain about malformed input. It is recommended to use instances as a context manager to ensure *frames* are properly finished.

If the size of the data being fed to this streaming compressor is known, you can declare it before compression begins:

```
>>> cctx = zstandard.ZstdCompressor()
>>> with cctx.stream_writer(fh, size=data_len) as compressor:
...     compressor.write(chunk0)
...     compressor.write(chunk1)
...     ...
```

Declaring the size of the source data allows compression parameters to be tuned. And if `write_content_size` is used, it also results in the content size being written into the frame header of the output data.

The size of chunks being `write()` to the destination can be specified:

```
>>> cctx = zstandard.ZstdCompressor()
>>> with cctx.stream_writer(fh, write_size=32768) as compressor:
...     ...
```

To see how much memory is being used by the streaming compressor:

```
>>> cctx = zstandard.ZstdCompressor()
>>> with cctx.stream_writer(fh) as compressor:
...     ...
...     byte_size = compressor.memory_size()
```

Thte total number of bytes written so far are exposed via `tell()`:

```
>>> cctx = zstandard.ZstdCompressor()
>>> with cctx.stream_writer(fh) as compressor:
...     ...
...     total_written = compressor.tell()
```

`stream_writer()` accepts a `write_return_read` boolean argument to control the return value of `write()`. When `False` (the default), `write()` returns the number of bytes that were `write()`'en to the underlying object. When `True`, `write()` returns the number of bytes read from the input that were subsequently written to the compressor. `True` is the *proper* behavior for `write()` as specified by the `io.RawIOBase` interface and will become the default value in a future release.

close()

closed

fileno()

flush(*flush_mode=0*)

Evict data from compressor's internal state and write it to inner stream.

Calling this method may result in 0 or more `write()` calls to the inner stream.

This method will also call `flush()` on the inner stream, if such a method exists.

Parameters **flush_mode** – How to flush the zstd compressor.

`zstandard.FLUSH_BLOCK` will flush data already sent to the compressor but not emitted to the inner stream. The stream is still writable after calling this. This is the default behavior.

See documentation for other `zstandard.FLUSH_*` constants for more flushing options.

Returns Integer number of bytes written to the inner stream.

isatty()

memory_size()

read(*size=-1*)

readable()

readall()

readinto(*b*)

readline(*size=-1*)

readlines(*hint=-1*)

seek(*offset*, *whence=None*)

seekable()

tell()

truncate(*size=None*)

writable()

write(*data*)
Send data to the compressor and possibly to the inner stream.

writelines(*lines*)

5.3 ZstdCompressionReader

class zstandard.**ZstdCompressionReader**(*compressor*, *source*, *read_size*, *closefd=True*)
Readable compressing stream wrapper.

`ZstdCompressionReader` is a read-only stream interface for obtaining compressed data from a source.

This type conforms to the `io.RawIOBase` interface and should be usable by any type that operates against a *file-object* (`typing.BinaryIO` in Python type hinting speak).

Instances are neither writable nor seekable (even if the underlying source is seekable). `readline()` and `readlines()` are not implemented because they don't make sense for compressed data. `tell()` returns the number of compressed bytes emitted so far.

Instances are obtained by calling *ZstdCompressor.stream_reader()*.

In this example, we open a file for reading and then wrap that file handle with a stream from which compressed data can be `read()`.

```
>>> with open(path, 'rb') as fh:
...     cctx = zstandard.ZstdCompressor()
...     reader = cctx.stream_reader(fh)
...     while True:
...         chunk = reader.read(16384)
...         if not chunk:
...             break
...
...         # Do something with compressed chunk.
```

Instances can also be used as context managers:

```
>>> with open(path, 'rb') as fh:
...     cctx = zstandard.ZstdCompressor()
...     with cctx.stream_reader(fh) as reader:
```

(continues on next page)

(continued from previous page)

```
...         while True:
...             chunk = reader.read(16384)
...             if not chunk:
...                 break
...
...             # Do something with compressed chunk.
```

When the context manager exits or `close()` is called, the stream is closed, underlying resources are released, and future operations against the compression stream will fail.

`stream_reader()` accepts a `size` argument specifying how large the input stream is. This is used to adjust compression parameters so they are tailored to the source size. e.g.

```
>>> with open(path, 'rb') as fh:
...     cctx = zstandard.ZstdCompressor()
...     with cctx.stream_reader(fh, size=os.stat(path).st_size) as reader:
...         ...
```

If the `source` is a stream, you can specify how large `read()` requests to that stream should be via the `read_size` argument. It defaults to `zstandard.COMPRESSION_RECOMMENDED_INPUT_SIZE`. e.g.

```
>>> with open(path, 'rb') as fh:
...     cctx = zstandard.ZstdCompressor()
...     # Will perform fh.read(8192) when obtaining data to feed into the
...     # compressor.
...     with cctx.stream_reader(fh, read_size=8192) as reader:
...         ...
```

close()

closed

flush()

isatty()

next()

read(*size=-1*)

read1(*size=-1*)

readable()

readall()

readinto(*b*)

readinto1(*b*)

readline()

readlines()

seekable()

tell()

writable()

write(*data*)

writelines(*ignored*)

5.4 ZstdCompressionObj

class zstandard.**ZstdCompressionObj**

A compressor conforming to the API in Python's standard library.

This type implements an API similar to compression types in Python's standard library such as `zlib.compressobj` and `bz2.BZ2Compressor`. This enables existing code targeting the standard library API to swap in this type to achieve zstd compression.

Important: The design of this API is not ideal for optimal performance.

The reason performance is not optimal is because the API is limited to returning a single buffer holding compressed data. When compressing data, we don't know how much data will be emitted. So in order to capture all this data in a single buffer, we need to perform buffer reallocations and/or extra memory copies. This can add significant overhead depending on the size or nature of the compressed data how much your application calls this type.

If performance is critical, consider an API like *ZstdCompressor.stream_reader()*, *ZstdCompressor.stream_writer()*, *ZstdCompressor.chunker()*, or *ZstdCompressor.read_to_iter()*, which result in less overhead managing buffers.

Instances are obtained by calling *ZstdCompressor.compressobj()*.

Here is how this API should be used:

```
>>> cctx = zstandard.ZstdCompressor()
>>> cobj = cctx.compressobj()
>>> data = cobj.compress(b"raw input 0")
>>> data = cobj.compress(b"raw input 1")
>>> data = cobj.flush()
```

Or to flush blocks:

```
>>> cctx.zstandard.ZstdCompressor()
>>> cobj = cctx.compressobj()
>>> data = cobj.compress(b"chunk in first block")
>>> data = cobj.flush(zstandard.COMPRESSOBJ_FLUSH_BLOCK)
>>> data = cobj.compress(b"chunk in second block")
>>> data = cobj.flush()
```

For best performance results, keep input chunks under 256KB. This avoids extra allocations for a large output object.

It is possible to declare the input size of the data that will be fed into the compressor:

```
>>> cctx = zstandard.ZstdCompressor()
>>> cobj = cctx.compressobj(size=6)
>>> data = cobj.compress(b"foobar")
>>> data = cobj.flush()
```

compress(*data*)

Send data to the compressor.

This method receives bytes to feed to the compressor and returns bytes constituting zstd compressed data.

The zstd compressor accumulates bytes and the returned bytes may be substantially smaller or larger than the size of the input data on any given call. The returned value may be the empty byte string (`b""`).

Parameters **data** – Data to write to the compressor.

Returns Compressed data.

flush (*flush_mode=0*)
Emit data accumulated in the compressor that hasn't been outputted yet.

The `flush_mode` argument controls how to end the stream.

`zstandard.COMPRESSOBJ_FLUSH_FINISH` (the default) ends the compression stream and finishes a zstd frame. Once this type of flush is performed, `compress()` and `flush()` can no longer be called. This type of flush **must** be called to end the compression context. If not called, the emitted data may be incomplete and may not be readable by a decompressor.

`zstandard.COMPRESSOBJ_FLUSH_BLOCK` will flush a zstd block. This ensures that all data fed to this instance will have been omitted and can be decoded by a decompressor. Flushes of this type can be performed multiple times. The next call to `compress()` will begin a new zstd block.

Parameters **flush_mode** – How to flush the zstd compressor.

Returns Compressed data.

5.5 ZstdCompressionChunker

class zstandard.**ZstdCompressionChunker**(*compressor*, *chunk_size*)
Compress data to uniformly sized chunks.

This type allows you to iteratively feed chunks of data into a compressor and produce output chunks of uniform size.

`compress()`, `flush()`, and `finish()` all return an iterator of `bytes` instances holding compressed data. The iterator may be empty. Callers MUST iterate through all elements of the returned iterator before performing another operation on the object or else the compressor's internal state may become confused. This can result in an exception being raised or malformed data being emitted.

All chunks emitted by `compress()` will have a length of the configured chunk size.

`flush()` and `finish()` may return a final chunk smaller than the configured chunk size.

Instances are obtained by calling *ZstdCompressor.chunker()*.

Here is how the API should be used:

```
>>> cctx = zstandard.ZstdCompressor()
>>> chunker = cctx.chunker(chunk_size=32768)
>>>
>>> with open(path, 'rb') as fh:
...     while True:
...         in_chunk = fh.read(32768)
...         if not in_chunk:
...             break
...
...         for out_chunk in chunker.compress(in_chunk):
...             # Do something with output chunk of size 32768.
...
...     for out_chunk in chunker.finish():
...         # Do something with output chunks that finalize the zstd frame.
```

This compressor type is often a better alternative to *ZstdCompressor.compressobj* because it has better performance properties.

`compressobj()` will emit output data as it is available. This results in a *stream* of output chunks of varying sizes. The consistency of the output chunk size with `chunker()` is more appropriate for many usages, such as sending compressed data to a socket.

`compressobj()` may also perform extra memory reallocations in order to dynamically adjust the sizes of the output chunks. Since `chunker()` output chunks are all the same size (except for flushed or final chunks), there is less memory allocation/copying overhead.

compress(*data*)
Feed new input data into the compressor.

Parameters **data** – Data to feed to compressor.

Returns Iterator of `bytes` representing chunks of compressed data.

finish()
Signals the end of input data.

No new data can be compressed after this method is called.

This method will flush buffered data and finish the zstd frame.

Returns Iterator of `bytes` of compressed data.

flush()
Flushes all data currently in the compressor.

Returns Iterator of `bytes` of compressed data.

CHAPTER 6

Decompression APIs

6.1 ZstdDecompressor

class zstandard.**ZstdDecompressor**(*dict_data=None*, *max_window_size=0*, *format=0*)

Context for performing zstandard decompression.

Each instance is essentially a wrapper around a ZSTD_DCtx from zstd's C API.

An instance can compress data various ways. Instances can be used multiple times.

The interface of this class is very similar to *zstandard.ZstdCompressor* (by design).

Assume that each ZstdDecompressor instance can only handle a single logical compression operation at the same time. i.e. if you call a method like decompressobj() to obtain multiple objects derived from the same ZstdDecompressor instance and attempt to use them simultaneously, errors will likely occur.

If you need to perform multiple logical decompression operations and you can't guarantee those operations are temporally non-overlapping, you need to obtain multiple ZstdDecompressor instances.

Unless specified otherwise, assume that no two methods of ZstdDecompressor instances can be called from multiple Python threads simultaneously. In other words, assume instances are not thread safe unless stated otherwise.

Parameters

- **dict_data** – Compression dictionary to use.
- **max_window_size** – Sets an upper limit on the window size for decompression operations in kibibytes. This setting can be used to prevent large memory allocations for inputs using large compression windows.
- **format** – Set the format of data for the decoder.

  By default this is zstandard.FORMAT_ZSTD1. It can be set to zstandard.FORMAT_ZSTD1_MAGICLESS to allow decoding frames without the 4 byte magic header. Not all decompression APIs support this mode.

copy_stream(*ifh*, *ofh*, *read_size=131075*, *write_size=131072*)
Copy data between streams, decompressing in the process.

Compressed data will be read from `ifh`, decompressed, and written to `ofh`.

```
>>> dctx = zstandard.ZstdDecompressor()
>>> dctx.copy_stream(ifh, ofh)
```

e.g. to decompress a file to another file:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> with open(input_path, 'rb') as ifh, open(output_path, 'wb') as ofh:
...     dctx.copy_stream(ifh, ofh)
```

The size of chunks being `read()` and `write()` from and to the streams can be specified:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> dctx.copy_stream(ifh, ofh, read_size=8192, write_size=16384)
```

Parameters

- **ifh** – Source stream to read compressed data from.

  Must have a `read()` method.

- **ofh** – Destination stream to write uncompressed data to.

  Must have a `write()` method.

- **read_size** – The number of bytes to `read()` from the source in a single operation.

- **write_size** – The number of bytes to `write()` to the destination in a single operation.

Returns 2-tuple of integers representing the number of bytes read and written, respectively.

decompress(*data*, *max_output_size=0*, *read_across_frames=False*, *allow_extra_data=True*)
Decompress data in a single operation.

This method will decompress the input data in a single operation and return the decompressed data.

The input bytes are expected to contain at least 1 full Zstandard frame (something compressed with *ZstdCompressor.compress()* or similar). If the input does not contain a full frame, an exception will be raised.

`read_across_frames` controls whether to read multiple zstandard frames in the input. When False, decompression stops after reading the first frame. This feature is not yet implemented but the argument is provided for forward API compatibility when the default is changed to True in a future release. For now, if you need to decompress multiple frames, use an API like *ZstdCompressor.stream_reader()* with `read_across_frames=True`.

`allow_extra_data` controls how to handle extra input data after a fully decoded frame. If False, any extra data (which could be a valid zstd frame) will result in `ZstdError` being raised. If True, extra data is silently ignored. The default will likely change to False in a future release when `read_across_frames` defaults to True.

If the input contains extra data after a full frame, that extra input data is silently ignored. This behavior is undesirable in many scenarios and will likely be changed or controllable in a future release (see #181).

If the frame header of the compressed data does not contain the content size, `max_output_size` must be specified or `ZstdError` will be raised. An allocation of size `max_output_size` will be performed and an attempt will be made to perform decompression into that buffer. If the buffer is too small or cannot be allocated, `ZstdError` will be raised. The buffer will be resized if it is too large.

Uncompressed data could be much larger than compressed data. As a result, calling this function could result in a very large memory allocation being performed to hold the uncompressed data. This could potentially result in `MemoryError` or system memory swapping. If you don't need the full output data in a single contiguous array in memory, consider using streaming decompression for more resilient memory behavior.

Usage:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> decompressed = dctx.decompress(data)
```

If the compressed data doesn't have its content size embedded within it, decompression can be attempted by specifying the `max_output_size` argument:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> uncompressed = dctx.decompress(data, max_output_size=1048576)
```

Ideally, `max_output_size` will be identical to the decompressed output size.

Important: If the exact size of decompressed data is unknown (not passed in explicitly and not stored in the zstd frame), for performance reasons it is encouraged to use a streaming API.

Parameters

- **`data`** – Compressed data to decompress.
- **`max_output_size`** – Integer max size of response.

  If `0`, there is no limit and we can attempt to allocate an output buffer of infinite size.

Returns `bytes` representing decompressed output.

`decompress_content_dict_chain` (*frames*)

Decompress a series of frames using the content dictionary chaining technique.

Such a list of frames is produced by compressing discrete inputs where each non-initial input is compressed with a *prefix* dictionary consisting of the content of the previous input.

For example, say you have the following inputs:

```
>>> inputs = [b"input 1", b"input 2", b"input 3"]
```

The zstd frame chain consists of:

1. `b"input 1"` compressed in standalone/discrete mode
2. `b"input 2"` compressed using `b"input 1"` as a *prefix* dictionary
3. `b"input 3"` compressed using `b"input 2"` as a *prefix* dictionary

Each zstd frame **must** have the content size written.

The following Python code can be used to produce a *prefix dictionary chain*:

```
>>> def make_chain(inputs):
...     frames = []
...
...     # First frame is compressed in standalone/discrete mode.
...     zctx = zstandard.ZstdCompressor()
```

(continues on next page)

(continued from previous page)

```
...     frames.append(zctx.compress(inputs[0]))
...
...     # Subsequent frames use the previous fulltext as a prefix dictionary
...     for i, raw in enumerate(inputs[1:]):
...         dict_data = zstandard.ZstdCompressionDict(
...             inputs[i], dict_type=zstandard.DICT_TYPE_RAWCONTENT)
...         zctx = zstandard.ZstdCompressor(dict_data=dict_data)
...         frames.append(zctx.compress(raw))
...
...     return frames
```

`decompress_content_dict_chain()` returns the uncompressed data of the last element in the input chain.

Note: It is possible to implement *prefix dictionary chain* decompression on top of other APIs. However, this function will likely be faster - especially for long input chains - as it avoids the overhead of instantiating and passing around intermediate objects between multiple functions.

Parameters **frames** – List of `bytes` holding compressed zstd frames.

Returns

decompressobj(*write_size=131072*)

Obtain a standard library compatible incremental decompressor.

See *ZstdDecompressionObj* for more documentation and usage examples.

Parameters **write_size** –

Returns *zstandard.ZstdDecompressionObj*

memory_size()

Size of decompression context, in bytes.

```
>>> dctx = zstandard.ZstdDecompressor()
>>> size = dctx.memory_size()
```

multi_decompress_to_buffer(*frames*, *decompressed_sizes=None*, *threads=0*)

Decompress multiple zstd frames to output buffers as a single operation.

(Experimental. Not available in CFFI backend.)

Compressed frames can be passed to the function as a `BufferWithSegments`, a `BufferWithSegmentsCollection`, or as a list containing objects that conform to the buffer protocol. For best performance, pass a `BufferWithSegmentsCollection` or a `BufferWithSegments`, as minimal input validation will be done for that type. If calling from Python (as opposed to C), constructing one of these instances may add overhead cancelling out the performance overhead of validation for list inputs.

Returns a `BufferWithSegmentsCollection` containing the decompressed data. All decompressed data is allocated in a single memory buffer. The `BufferWithSegments` instance tracks which objects are at which offsets and their respective lengths.

```
>>> dctx = zstandard.ZstdDecompressor()
>>> results = dctx.multi_decompress_to_buffer([b'...', b'...'])
```

The decompressed size of each frame MUST be discoverable. It can either be embedded within the zstd frame or passed in via the `decompressed_sizes` argument.

The `decompressed_sizes` argument is an object conforming to the buffer protocol which holds an array of 64-bit unsigned integers in the machine's native format defining the decompressed sizes of each frame. If this argument is passed, it avoids having to scan each frame for its decompressed size. This frame scanning can add noticeable overhead in some scenarios.

```
>>> frames = [...]
>>> sizes = struct.pack('=QQQQ', len0, len1, len2, len3)
>>>
>>> dctx = zstandard.ZstdDecompressor()
>>> results = dctx.multi_decompress_to_buffer(frames, decompressed_
↪sizes=sizes)
```

Note: It is possible to pass a `mmap.mmap()` instance into this function by wrapping it with a `BufferWithSegments` instance (which will define the offsets of frames within the memory mapped region).

This function is logically equivalent to performing `ZstdCompressor.decompress()` on each input frame and returning the result.

This function exists to perform decompression on multiple frames as fast as possible by having as little overhead as possible. Since decompression is performed as a single operation and since the decompressed output is stored in a single buffer, extra memory allocations, Python objects, and Python function calls are avoided. This is ideal for scenarios where callers know up front that they need to access data for multiple frames, such as when *delta chains* are being used.

Currently, the implementation always spawns multiple threads when requested, even if the amount of work to do is small. In the future, it will be smarter about avoiding threads and their associated overhead when the amount of work to do is small.

Parameters

- **frames** – Source defining zstd frames to decompress.
- **decompressed_sizes** – Array of integers representing sizes of decompressed zstd frames.
- **threads** – How many threads to use for decompression operations.

  Negative values will use the same number of threads as logical CPUs on the machine. Values `0` or `1` use a single thread.

Returns `BufferWithSegmentsCollection`

read_to_iter(*reader*, *read_size=131075*, *write_size=131072*, *skip_bytes=0*)

Read compressed data to an iterator of uncompressed chunks.

This method will read data from `reader`, feed it to a decompressor, and emit `bytes` chunks representing the decompressed result.

```
>>> dctx = zstandard.ZstdDecompressor()
>>> for chunk in dctx.read_to_iter(fh):
...     # Do something with original data.
```

`read_to_iter()` accepts an object with a `read(size)` method that will return compressed bytes or an object conforming to the buffer protocol.

`read_to_iter()` returns an iterator whose elements are chunks of the decompressed data.

The size of requested `read()` from the source can be specified:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> for chunk in dctx.read_to_iter(fh, read_size=16384):
...     pass
```

It is also possible to skip leading bytes in the input data:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> for chunk in dctx.read_to_iter(fh, skip_bytes=1):
...     pass
```

Tip: Skipping leading bytes is useful if the source data contains extra *header* data. Traditionally, you would need to create a slice or `memoryview` of the data you want to decompress. This would create overhead. It is more efficient to pass the offset into this API.

Similarly to *ZstdCompressor.read_to_iter()*, the consumer of the iterator controls when data is decompressed. If the iterator isn't consumed, decompression is put on hold.

When `read_to_iter()` is passed an object conforming to the buffer protocol, the behavior may seem similar to what occurs when the simple decompression API is used. However, this API works when the decompressed size is unknown. Furthermore, if feeding large inputs, the decompressor will work in chunks instead of performing a single operation.

Parameters

- **reader** – Source of compressed data. Can be any object with a `read(size)` method or any object conforming to the buffer protocol.
- **read_size** – Integer size of data chunks to read from `reader` and feed into the decompressor.
- **write_size** – Integer size of data chunks to emit from iterator.
- **skip_bytes** – Integer number of bytes to skip over before sending data into the decompressor.

Returns Iterator of `bytes` representing uncompressed data.

stream_reader(*source*, *read_size=131075*, *read_across_frames=False*, *closefd=True*)

Read-only stream wrapper that performs decompression.

This method obtains an object that conforms to the `io.RawIOBase` interface and performs transparent decompression via `read()` operations. Source data is obtained by calling `read()` on a source stream or object implementing the buffer protocol.

See *zstandard.ZstdDecompressionReader* for more documentation and usage examples.

Parameters

- **source** – Source of compressed data to decompress. Can be any object with a `read(size)` method or that conforms to the buffer protocol.
- **read_size** – Integer number of bytes to read from the source and feed into the compressor at a time.
- **read_across_frames** – Whether to read data across multiple zstd frames. If False, decompression is stopped at frame boundaries.
- **closefd** – Whether to close the source stream when this instance is closed.

Returns *zstandard.ZstdDecompressionReader*.

stream_writer(*writer*, *write_size=131072*, *write_return_read=True*, *closefd=True*)
Push-based stream wrapper that performs decompression.

This method constructs a stream wrapper that conforms to the `io.RawIOBase` interface and performs transparent decompression when writing to a wrapper stream.

See *zstandard.ZstdDecompressionWriter* for more documentation and usage examples.

Parameters

- **writer** – Destination for decompressed output. Can be any object with a `write(data)`.
- **write_size** – Integer size of chunks to `write()` to `writer`.
- **write_return_read** – Whether `write()` should return the number of bytes of input consumed. If False, `write()` returns the number of bytes sent to the inner stream.
- **closefd** – Whether to `close()` the inner stream when this stream is closed.

Returns *zstandard.ZstdDecompressionWriter*

6.2 ZstdDecompressionWriter

class zstandard.**ZstdDecompressionWriter**(*decompressor*, *writer*, *write_size*, *write_return_read*, *closefd=True*)
Write-only stream wrapper that performs decompression.

This type provides a writable stream that performs decompression and writes decompressed data to another stream.

This type implements the `io.RawIOBase` interface. Only methods that involve writing will do useful things.

Behavior is similar to *ZstdCompressor.stream_writer()*: compressed data is sent to the decompressor by calling `write(data)` and decompressed output is written to the inner stream by calling its `write(data)` method:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> decompressor = dctx.stream_writer(fh)
>>> # Will call fh.write() with uncompressed data.
>>> decompressor.write(compressed_data)
```

Instances can be used as context managers. However, context managers add no extra special behavior other than automatically calling `close()` when they exit.

Calling `close()` will mark the stream as closed and subsequent I/O operations will raise `ValueError` (per the documented behavior of `io.RawIOBase`). `close()` will also call `close()` on the underlying stream if such a method exists and the instance was created with `closefd=True`.

The size of chunks to `write()` to the destination can be specified:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> with dctx.stream_writer(fh, write_size=16384) as decompressor:
>>>     pass
```

You can see how much memory is being used by the decompressor:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> with dctx.stream_writer(fh) as decompressor:
>>>     byte_size = decompressor.memory_size()
```

`stream_writer()` accepts a `write_return_read` boolean argument to control the return value of `write()`. When `True` (the default)‘‘, `write()` returns the number of bytes that were read from the input. When `False`, `write()` returns the number of bytes that were `write()` to the inner stream.

close()

closed

fileno()

flush()

isatty()

memory_size()

read(*size=-1*)

readable()

readall()

readinto(*b*)

readline(*size=-1*)

readlines(*hint=-1*)

seek(*offset*, *whence=None*)

seekable()

tell()

truncate(*size=None*)

writable()

write(*data*)

writelines(*lines*)

6.3 ZstdDecompressionReader

class zstandard.**ZstdDecompressionReader**(*decompressor*, *source*, *read_size*, *read_across_frames*, *closefd=True*)

Read only decompressor that pull uncompressed data from another stream.

This type provides a read-only stream interface for performing transparent decompression from another stream or data source. It conforms to the `io.RawIOBase` interface. Only methods relevant to reading are implemented.

```
>>> with open(path, 'rb') as fh:
>>> dctx = zstandard.ZstdDecompressor()
>>> reader = dctx.stream_reader(fh)
>>> while True:
...     chunk = reader.read(16384)
...     if not chunk:
```

(continues on next page)

(continued from previous page)

```
...             break
...         # Do something with decompressed chunk.
```

The stream can also be used as a context manager:

```
>>> with open(path, 'rb') as fh:
...     dctx = zstandard.ZstdDecompressor()
...     with dctx.stream_reader(fh) as reader:
...         ...
```

When used as a context manager, the stream is closed and the underlying resources are released when the context manager exits. Future operations against the stream will fail.

The `source` argument to `stream_reader()` can be any object with a `read(size)` method or any object implementing the *buffer protocol*.

If the `source` is a stream, you can specify how large `read()` requests to that stream should be via the `read_size` argument. It defaults to `zstandard.DECOMPRESSION_RECOMMENDED_INPUT_SIZE`.:

```
>>> with open(path, 'rb') as fh:
...     dctx = zstandard.ZstdDecompressor()
...     # Will perform fh.read(8192) when obtaining data for the decompressor.
...     with dctx.stream_reader(fh, read_size=8192) as reader:
...         ...
```

Instances are *partially* seekable. Absolute and relative positions (`SEEK_SET` and `SEEK_CUR`) forward of the current position are allowed. Offsets behind the current read position and offsets relative to the end of stream are not allowed and will raise `ValueError` if attempted.

`tell()` returns the number of decompressed bytes read so far.

Not all I/O methods are implemented. Notably missing is support for `readline()`, `readlines()`, and linewise iteration support. This is because streams operate on binary data - not text data. If you want to convert decompressed output to text, you can chain an `io.TextIOWrapper` to the stream:

```
>>> with open(path, 'rb') as fh:
...     dctx = zstandard.ZstdDecompressor()
...     stream_reader = dctx.stream_reader(fh)
...     text_stream = io.TextIOWrapper(stream_reader, encoding='utf-8')
...     for line in text_stream:
...         ...
```

close()

closed

flush()

isatty()

next()

read(*size=-1*)

read1(*size=-1*)

readable()

readall()

readinto(*b*)

readinto1(*b*)

readline(*size=-1*)

readlines(*hint=-1*)

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

seekable()

tell()

writable()

write(*data*)

writelines(*lines*)

6.4 ZstdDecompressionObj

class zstandard.**ZstdDecompressionObj**(*decompressor*, *write_size*)

A standard library API compatible decompressor.

This type implements a compressor that conforms to the API by other decompressors in Python's standard library. e.g. `zlib.decompressobj` or `bz2.BZ2Decompressor`. This allows callers to use zstd compression while conforming to a similar API.

Compressed data chunks are fed into `decompress(data)` and uncompressed output (or an empty bytes) is returned. Output from subsequent calls needs to be concatenated to reassemble the full decompressed byte sequence.

Each instance is single use: once an input frame is decoded, `decompress()` can no longer be called.

```
>>> dctx = zstandard.ZstdDecompressor()
>>> dobj = dctx.decompressobj()
>>> data = dobj.decompress(compressed_chunk_0)
>>> data = dobj.decompress(compressed_chunk_1)
```

By default, calls to `decompress()` write output data in chunks of size `DECOMPRESSION_RECOMMENDED_OUTPUT_SIZE`. These chunks are concatenated before being returned to the caller. It is possible to define the size of these temporary chunks by passing `write_size` to `decompressobj()`:

```
>>> dctx = zstandard.ZstdDecompressor()
>>> dobj = dctx.decompressobj(write_size=1048576)
```

Note: Because calls to `decompress()` may need to perform multiple memory (re)allocations, this streaming decompression API isn't as efficient as other APIs.

---

decompress(*data*)

Send compressed data to the decompressor and obtain decompressed data.

Parameters **data** – Data to feed into the decompressor.

Returns Decompressed bytes.

eof

Whether the end of the compressed data stream has been reached.

flush(*length=0*)
Effectively a no-op.

Implemented for compatibility with the standard library APIs.

Safe to call at any time.

Returns Empty bytes.

unconsumed_tail
Data that has not yet been fed into the decompressor.

unused_data
Bytes past the end of compressed data.

If `decompress()` is fed additional data beyond the end of a zstd frame, this value will be non-empty once `decompress()` fully decodes the input frame.

CHAPTER 7

Multi-Threaded Compression

`ZstdCompressor` accepts a `threads` argument that controls the number of threads to use for compression. The way this works is that input is split into segments and each segment is fed into a worker pool for compression. Once a segment is compressed, it is flushed/appended to the output.

Note: These threads are created at the C layer and are not Python threads. So they work outside the GIL. It is therefore possible to CPU saturate multiple cores from Python.

The segment size for multi-threaded compression is chosen from the window size of the compressor. This is derived from the `window_log` attribute of a `ZstdCompressionParameters` instance. By default, segment sizes are in the 1+MB range.

If multi-threaded compression is requested and the input is smaller than the configured segment size, only a single compression thread will be used. If the input is smaller than the segment size multiplied by the thread pool size or if data cannot be delivered to the compressor fast enough, not all requested compressor threads may be active simultaneously.

Compared to non-multi-threaded compression, multi-threaded compression has higher per-operation overhead. This includes extra memory operations, thread creation, lock acquisition, etc.

Due to the nature of multi-threaded compression using *N* compression *states*, the output from multi-threaded compression will likely be larger than non-multi-threaded compression. The difference is usually small. But there is a CPU/wall time versus size trade off that may warrant investigation.

Output from multi-threaded compression does not require any special handling on the decompression side. To the decompressor, data generated with single threaded compressor looks the same as data generated by a multi-threaded compressor and does not require any special handling or additional resource requirements.

CHAPTER 8

Dictionaries

8.1 ZstdCompressionDict

class zstandard.**ZstdCompressionDict**(*data*, *dict_type=0*, *k=0*, *d=0*)

Represents a computed compression dictionary.

Instances are obtained by calling *train_dictionary()* or by passing bytes obtained from another source into the constructor.

Instances can be constructed from bytes:

```
>>> dict_data = zstandard.ZstdCompressionDict(data)
```

It is possible to construct a dictionary from *any* data. If the data doesn't begin with a magic header, it will be treated as a *prefix* dictionary. *Prefix* dictionaries allow compression operations to reference raw data within the dictionary.

It is possible to force the use of *prefix* dictionaries or to require a dictionary header:

```
>>> dict_data = zstandard.ZstdCompressionDict(data, dict_type=zstandard.DICT_TYPE_
↪RAWCONTENT)
>>> dict_data = zstandard.ZstdCompressionDict(data, dict_type=zstandard.DICT_TYPE_
↪FULLDICT)
```

You can see how many bytes are in the dictionary by calling `len()`:

```
>>> dict_data = zstandard.train_dictionary(size, samples)
>>> dict_size = len(dict_data)  # will not be larger than ``size``
```

Once you have a dictionary, you can pass it to the objects performing compression and decompression:

```
>>> dict_data = zstandard.train_dictionary(131072, samples)
>>> cctx = zstandard.ZstdCompressor(dict_data=dict_data)
>>> for source_data in input_data:
...     compressed = cctx.compress(source_data)
```

(continues on next page)

(continued from previous page)

```
...     # Do something with compressed data.
...
>>> dctx = zstandard.ZstdDecompressor(dict_data=dict_data)
>>> for compressed_data in input_data:
...     buffer = io.BytesIO()
...     with dctx.stream_writer(buffer) as decompressor:
...         decompressor.write(compressed_data)
...         # Do something with raw data in ``buffer``.
```

Dictionaries have unique integer IDs. You can retrieve this ID via:

```
>>> dict_id = zstandard.dictionary_id(dict_data)
```

You can obtain the raw data in the dict (useful for persisting and constructing a `ZstdCompressionDict` later) via `as_bytes()`:

```
>>> dict_data = zstandard.train_dictionary(size, samples)
>>> raw_data = dict_data.as_bytes()
```

By default, when a `ZstdCompressionDict` is *attached* to a `ZstdCompressor`, each `ZstdCompressor` performs work to prepare the dictionary for use. This is fine if only 1 compression operation is being performed or if the `ZstdCompressor` is being reused for multiple operations. But if multiple `ZstdCompressor` instances are being used with the dictionary, this can add overhead.

It is possible to *precompute* the dictionary so it can readily be consumed by multiple `ZstdCompressor` instances:

```
>>> d = zstandard.ZstdCompressionDict(data)
>>> # Precompute for compression level 3.
>>> d.precompute_compress(level=3)
>>> # Precompute with specific compression parameters.
>>> params = zstandard.ZstdCompressionParameters(...)
>>> d.precompute_compress(compression_params=params)
```

Note: When a dictionary is precomputed, the compression parameters used to precompute the dictionary overwrite some of the compression parameters specified to `ZstdCompressor`.

Parameters

- **data** – Dictionary data.
- **dict_type** – Type of dictionary. One of the `DICT_TYPE_*` constants.

as_bytes()
Obtain the `bytes` representation of the dictionary.

dict_id()
Obtain the integer ID of the dictionary.

precompute_compress(*level=0*, *compression_params=None*)
Precompute a dictionary os it can be used by multiple compressors.

Calling this method on an instance that will be used by multiple *ZstdCompressor* instances will improve performance.

8.2 Training Dictionaries

Unless using *prefix* dictionaries, dictionary data is produced by *training* on existing data using the `train_dictionary()` function.

`zstandard.`**`train_dictionary`**(*dict_size*, *samples*, *k=0*, *d=0*, *f=0*, *split_point=0.0*, *accel=0*, *notifications=0*, *dict_id=0*, *level=0*, *steps=0*, *threads=0*)

Train a dictionary from sample data using the COVER algorithm.

A compression dictionary of size `dict_size` will be created from the iterable of `samples`. The raw dictionary bytes will be returned.

The dictionary training mechanism is known as *cover*. More details about it are available in the paper *Effective Construction of Relative Lempel-Ziv Dictionaries* (authors: Liao, Petri, Moffat, Wirth).

The cover algorithm takes parameters `k` and `d`. These are the *segment size* and *dmer size*, respectively. The returned dictionary instance created by this function has `k` and `d` attributes containing the values for these parameters. If a `ZstdCompressionDict` is constructed from raw bytes data (a content-only dictionary), the `k` and `d` attributes will be `0`.

The segment and dmer size parameters to the cover algorithm can either be specified manually or `train_dictionary()` can try multiple values and pick the best one, where *best* means the smallest compressed data size. This later mode is called *optimization* mode.

Under the hood, this function always calls `ZDICT_optimizeTrainFromBuffer_fastCover()`. See the corresponding C library documentation for more.

If neither `steps` nor `threads` is defined, defaults for `d`, `steps`, and `level` will be used that are equivalent with what `ZDICT_trainFromBuffer()` would use.

Parameters

- **`dict_size`** – Target size in bytes of the dictionary to generate.
- **`samples`** – A list of bytes holding samples the dictionary will be trained from.
- **`k`** – Segment size : constraint: 0 < k : Reasonable range [16, 2048+]
- **`d`** – dmer size : constraint: 0 < d <= k : Reasonable range [6, 16]
- **`f`** – log of size of frequency array : constraint: 0 < f <= 31 : 1 means default(20)
- **`split_point`** – Percentage of samples used for training: Only used for optimization. The first # samples * `split_point` samples will be used to training. The last # samples * (1 - split_point) samples will be used for testing. 0 means default (0.75), 1.0 when all samples are used for both training and testing.
- **`accel`** – Acceleration level: constraint: 0 < accel <= 10. Higher means faster and less accurate, 0 means default(1).
- **`dict_id`** – Integer dictionary ID for the produced dictionary. Default is 0, which uses a random value.
- **`steps`** – Number of steps through `k` values to perform when trying parameter variations.
- **`threads`** – Number of threads to use when trying parameter variations. Default is 0, which means to use a single thread. A negative value can be specified to use as many threads as there are detected logical CPUs.
- **`level`** – Integer target compression level when trying parameter variations.

- **notifications** – Controls writing of informational messages to `stderr`. `0` (the default) means to write nothing. `1` writes errors. `2` writes progression info. `3` writes more details. And `4` writes all info.

CHAPTER 9

ZstdCompressionParameters

Zstandard offers a high-level *compression level* that maps to lower-level compression parameters. For many consumers, this numeric level is the only compression setting you'll need to touch.

But for advanced use cases, it might be desirable to tweak these lower-level settings.

The `ZstdCompressionParameters` type represents these low-level compression settings.

class zstandard.**ZstdCompressionParameters**(*format=0*, *compression_level=0*, *window_log=0*, *hash_log=0*, *chain_log=0*, *search_log=0*, *min_match=0*, *target_length=0*, *strategy=-1*, *write_content_size=1*, *write_checksum=0*, *write_dict_id=0*, *job_size=0*, *overlap_log=-1*, *force_max_window=0*, *enable_ldm=0*, *ldm_hash_log=0*, *ldm_min_match=0*, *ldm_bucket_size_log=0*, *ldm_hash_rate_log=-1*, *threads=0*)

Low-level zstd compression parameters.

This type represents a collection of parameters to control how zstd compression is performed.

Instances can be constructed from raw parameters or derived from a base set of defaults specified from a compression level (recommended) via *`ZstdCompressionParameters.from_level()`*.

```
>>> # Derive compression settings for compression level 7.
>>> params = zstandard.ZstdCompressionParameters.from_level(7)
```

```
>>> # With an input size of 1MB
>>> params = zstandard.ZstdCompressionParameters.from_level(7, source_
↪size=1048576)
```

Using `from_level()`, it is also possible to override individual compression parameters or to define additional settings that aren't automatically derived. e.g.:

```
>>> params = zstandard.ZstdCompressionParameters.from_level(4, window_log=10)
>>> params = zstandard.ZstdCompressionParameters.from_level(5, threads=4)
```

Or you can define low-level compression settings directly:

```
>>> params = zstandard.ZstdCompressionParameters(window_log=12, enable_ldm=True)
```

Once a `ZstdCompressionParameters` instance is obtained, it can be used to configure a compressor:

```
>>> cctx = zstandard.ZstdCompressor(compression_params=params)
```

Some of these are very low-level settings. It may help to consult the official zstandard documentation for their behavior. Look for the `ZSTD_p_*` constants in `zstd.h` (https://github.com/facebook/zstd/blob/dev/lib/zstd.h).

chain_log

compression_level

enable_ldm

estimated_compression_context_size()
Estimated size in bytes needed to compress with these parameters.

force_max_window

format

static from_level(*level*, *source_size=0*, *dict_size=0*, ***kwargs*)
Create compression parameters from a compression level.

Parameters

- **level** – Integer compression level.
- **source_size** – Integer size in bytes of source to be compressed.
- **dict_size** – Integer size in bytes of compression dictionary to use.

Returns *ZstdCompressionParameters*

hash_log

job_size

ldm_bucket_size_log

ldm_hash_log

ldm_hash_rate_log

ldm_min_match

min_match

overlap_log

search_log

strategy

target_length

threads

window_log

write_checksum

write_content_size

write_dict_id

CHAPTER 10

Miscellaneous APIs

10.1 Frame Inspection

Data emitted from zstd compression is encapsulated in a *frame*. This frame begins with a 4 byte *magic number* header followed by 2 to 14 bytes describing the frame in more detail. For more info, see https://github.com/facebook/zstd/blob/master/doc/zstd_compression_format.md.

`zstandard.`**`get_frame_parameters`**(*data*)
Parse a zstd frame header into frame parameters.

Depending on which fields are present in the frame and their values, the length of the frame parameters varies. If insufficient bytes are passed in to fully parse the frame parameters, `ZstdError` is raised. To ensure frame parameters can be parsed, pass in at least 18 bytes.

Parameters **`data`** – Data from which to read frame parameters.

Returns *`FrameParameters`*

`zstandard.`**`frame_header_size`**(*data*)
Obtain the size of a frame header.

Returns Integer size in bytes.

`zstandard.`**`frame_content_size`**(*data*)
Obtain the decompressed size of a frame.

The returned value is usually accurate. But strictly speaking it should not be trusted.

Returns `-1` if size unknown and a non-negative integer otherwise.

`class` `zstandard.`**`FrameParameters`**(*fparams*)
Information about a zstd frame.

Instances have the following attributes:

`content_size` Integer size of original, uncompressed content. This will be `0` if the original content size isn't written to the frame (controlled with the `write_content_size` argument to `ZstdCompressor`) or if the input content size was `0`.

window_size Integer size of maximum back-reference distance in compressed data.

dict_id Integer of dictionary ID used for compression. `0` if no dictionary ID was used or if the dictionary ID was `0`.

has_checksum Bool indicating whether a 4 byte content checksum is stored at the end of the frame.

10.2 estimate_decompression_context_size()

zstandard.**estimate_decompression_context_size**()
Estimate the memory size requirements for a decompressor instance.

Returns Integer number of bytes.

10.3 open()

zstandard.**open**(*filename*, *mode='rb'*, *cctx=None*, *dctx=None*, *encoding=None*, *errors=None*, *newline=None*, *closefd=None*)
Create a file object with zstd (de)compression.

The object returned from this function will be a *ZstdDecompressionReader* if opened for reading in binary mode, a *ZstdCompressionWriter* if opened for writing in binary mode, or an `io.TextIOWrapper` if opened for reading or writing in text mode.

Parameters

- **filename** – `bytes`, `str`, or `os.PathLike` defining a file to open or a file object (with a `read()` or `write()` method).
- **mode** – `str` File open mode. Accepts any of the open modes recognized by `open()`.
- **cctx** – `ZstdCompressor` to use for compression. If not specified and file is opened for writing, the default `ZstdCompressor` will be used.
- **dctx** – `ZstdDecompressor` to use for decompression. If not specified and file is opened for reading, the default `ZstdDecompressor` will be used.
- **encoding** – `str` that defines text encoding to use when file is opened in text mode.
- **errors** – `str` defining text encoding error handling mode.
- **newline** – `str` defining newline to use in text mode.
- **closefd** –

  `bool` **whether to close the file when the returned object is closed.** Only used if a file object is passed. If a filename is specified, the opened file is always closed when the returned object is closed.

10.4 compress()

zstandard.**compress**(*data: ByteString*, *level: int = 3*) → bytes
Compress source data using the zstd compression format.

This performs one-shot compression using basic/default compression settings.

This method is provided for convenience and is equivalent to calling `ZstdCompressor(level=level).compress(data)`.

If you find yourself calling this function in a tight loop, performance will be greater if you construct a single `ZstdCompressor` and repeatedly call `compress()` on it.

10.5 decompress()

zstandard.**decompress**(*data: ByteString*, *max_output_size: int = 0*) → bytes

Decompress a zstd frame into its original data.

This performs one-shot decompression using basic/default compression settings.

This method is provided for convenience and is equivalent to calling `ZstdDecompressor().decompress(data, max_output_size=max_output_size)`.

If you find yourself calling this function in a tight loop, performance will be greater if you construct a single `ZstdDecompressor` and repeatedly call `decompress()` on it.

10.6 Constants

The following module constants/attributes are exposed:

ZSTD_VERSION This module attribute exposes a 3-tuple of the Zstandard version. e.g. `(1, 0, 0)`

MAX_COMPRESSION_LEVEL Integer max compression level accepted by compression functions

COMPRESSION_RECOMMENDED_INPUT_SIZE Recommended chunk size to feed to compressor functions

COMPRESSION_RECOMMENDED_OUTPUT_SIZE Recommended chunk size for compression output

DECOMPRESSION_RECOMMENDED_INPUT_SIZE Recommended chunk size to feed into decompresor functions

DECOMPRESSION_RECOMMENDED_OUTPUT_SIZE Recommended chunk size for decompression output

FRAME_HEADER bytes containing header of the Zstandard frame

MAGIC_NUMBER Frame header as an integer

FLUSH_BLOCK Flushing behavior that denotes to flush a zstd block. A decompressor will be able to decode all data fed into the compressor so far.

FLUSH_FRAME Flushing behavior that denotes to end a zstd frame. Any new data fed to the compressor will start a new frame.

CONTENTSIZE_UNKNOWN Value for content size when the content size is unknown.

CONTENTSIZE_ERROR Value for content size when content size couldn't be determined.

WINDOWLOG_MIN Minimum value for compression parameter

WINDOWLOG_MAX Maximum value for compression parameter

CHAINLOG_MIN Minimum value for compression parameter

CHAINLOG_MAX Maximum value for compression parameter

HASHLOG_MIN Minimum value for compression parameter

HASHLOG_MAX Maximum value for compression parameter

SEARCHLOG_MIN Minimum value for compression parameter

SEARCHLOG_MAX Maximum value for compression parameter

MINMATCH_MIN Minimum value for compression parameter

MINMATCH_MAX Maximum value for compression parameter

SEARCHLENGTH_MIN Minimum value for compression parameter

Deprecated: use `MINMATCH_MIN`

SEARCHLENGTH_MAX Maximum value for compression parameter

Deprecated: use `MINMATCH_MAX`

TARGETLENGTH_MIN Minimum value for compression parameter

STRATEGY_FAST Compression strategy

STRATEGY_DFAST Compression strategy

STRATEGY_GREEDY Compression strategy

STRATEGY_LAZY Compression strategy

STRATEGY_LAZY2 Compression strategy

STRATEGY_BTLAZY2 Compression strategy

STRATEGY_BTOPT Compression strategy

STRATEGY_BTULTRA Compression strategy

STRATEGY_BTULTRA2 Compression strategy

FORMAT_ZSTD1 Zstandard frame format

FORMAT_ZSTD1_MAGICLESS Zstandard frame format without magic header

CHAPTER 11

Buffer Types

The `zstandard` module exposes a handful of custom types for interfacing with memory buffers. The primary goal of these types is to facilitate efficient multi-object operations.

The essential idea is to have a single memory allocation provide backing storage for multiple logical objects. This has 2 main advantages: fewer allocations and optimal memory access patterns. This avoids having to allocate a Python object for each logical object and furthermore ensures that access of data for objects can be sequential (read: fast) in memory.

11.1 BufferSegment

class zstandard.**BufferSegment**

Represents a segment within a `BufferWithSegments`.

This type is essentially a reference to N bytes within a `BufferWithSegments`.

The object conforms to the buffer protocol.

offset

The byte offset of this segment within its parent buffer.

tobytes()

Obtain bytes copy of this segment.

11.2 BufferSegments

class zstandard.**BufferSegments**

Represents an array of `(offset, length)` integers.

This type is effectively an index used by *`BufferWithSegments`*.

The array members are 64-bit unsigned integers using host/native bit order.

Instances conform to the buffer protocol.

11.3 BufferWithSegments

class zstandard.**BufferWithSegments**

A memory buffer containing N discrete items of known lengths.

This type is essentially a fixed size memory address and an array of 2-tuples of `(offset, length)` 64-bit unsigned native-endian integers defining the byte offset and length of each segment within the buffer.

Instances behave like containers.

Instances also conform to the buffer protocol. So a reference to the backing bytes can be obtained via `memoryview(o)`. A *copy* of the backing bytes can be obtained via `.tobytes()`.

This type exists to facilitate operations against N>1 items without the overhead of Python object creation and management. Used with APIs like *ZstdDecompressor.multi_decompress_to_buffer()*, it is possible to decompress many objects in parallel without the GIL held, leading to even better performance.

segments()

Obtain the array of `(offset, length)` segments in the buffer.

Returns *BufferSegments*

size

Total sizein bytes of the backing buffer.

tobytes()

Obtain bytes copy of this instance.

11.4 BufferWithSegmentsCollection

class zstandard.**BufferWithSegmentsCollection**

A virtual spanning view over multiple BufferWithSegments.

Instances are constructed from 1 or more *BufferWithSegments* instances. The resulting object behaves like an ordered sequence whose members are the segments within each `BufferWithSegments`.

If the object is composed of 2 `BufferWithSegments` instances with the first having 2 segments and the second have 3 segments, then `b[0]` and `b[1]` access segments in the first object and `b[2]`, `b[3]`, and `b[4]` access segments from the second.

CHAPTER 12

Contributing

Once you have the source code, the extension can be built via setup.py:

```
$ python setup.py build_ext
```

We recommend testing with `pytest`:

```
$ pytest
```

Tests use the `hypothesis` Python package to perform fuzzing. If you don't have it, those tests won't run. Since the fuzzing tests take longer to execute than normal tests, you'll need to opt in to running them by setting the `ZSTD_SLOW_TESTS` environment variable.

The `cffi` Python package needs to be installed in order to build the CFFI bindings. If it isn't present, the CFFI bindings won't be built.

To create a virtualenv with all development dependencies, do something like the following:

```
$ python3 -m venv venv

$ source venv/bin/activate
$ pip install -r ci/requirements.txt
```

CHAPTER 13

Version History

13.1 1.0.0 (not yet released)

13.1.1 Actions Blocking Release

- Properly handle non-blocking I/O and partial writes for objects implementing `io.RawIOBase`.
- Consider making reads across frames configurable behavior.
- Overall API design review.
- Use Python allocator where possible.
- Figure out what to do about experimental APIs not implemented by CFFI.
- APIs for auto adjusting compression parameters based on input size. e.g. clamping the window log so it isn't too large for input.
- Consider allowing compressor and decompressor instances to be thread safe, support concurrent operations. Or track when an operation is in progress and refuse to let concurrent operations use the same instance.
- Support for magic-less frames for all decompression operations (`decompress()` doesn't work due to sniffing the content size and the lack of a ZSTD API to sniff magic-less frames - this should be fixed in 1.3.5.).
- Audit for complete flushing when ending compression streams.
- Deprecate legacy APIs.
- Audit for ability to control read/write sizes on all APIs.
- Detect memory leaks via bench.py.
- Remove low-level compression parameters from `ZstdCompressor.__init__` and require use of `ZstdCompressionParameters`.
- Expose `ZSTD_getFrameProgression()` from more compressor types.
- Support modifying compression parameters mid operation when supported by zstd API.

- Expose `ZSTD_CLEVEL_DEFAULT` constant.
- Expose `ZSTD_SRCSIZEHINT_{MIN,MAX}` constants.
- Support `ZSTD_p_forceAttachDict` compression parameter.
- Support `ZSTD_dictForceLoad` dictionary compression parameter.
- Support `ZSTD_c_targetCBlockSize` compression parameter.
- Support `ZSTD_c_literalCompressionMode` compression parameter.
- Support `ZSTD_c_srcSizeHint` compression parameter.
- Use `ZSTD_CCtx_getParameter()`/`ZSTD_CCtxParam_getParameter()` for retrieving compression parameters.
- Consider exposing `ZSTDMT_toFlushNow()`.
- Expose `ZSTD_Sequence` struct and related `ZSTD_getSequences()` API.
- Expose and enforce `ZSTD_minCLevel()` for minimum compression level.
- Consider a `chunker()` API for decompression.
- Consider stats for `chunker()` API, including finding the last consumed offset of input data.
- Consider exposing `ZSTD_cParam_getBounds()` and `ZSTD_dParam_getBounds()` APIs.
- Consider controls over resetting compression contexts (session only, parameters, or session and parameters).
- Consider exposing `ZSTD_d_stableOutBuffer`.
- Support `ZSTD_c_enableDedicatedDictSearch`.
- Support `ZSTD_c_stableInBuffer`.
- Support `ZSTD_c_stableOutBuffer`.
- Support `ZSTD_c_blockDelimiters`.
- Support `ZSTD_c_validateSequences`.
- Support `ZSTD_c_useBlockSplitter`.
- Support `ZSTD_c_useRowMatchFinder`.
- Support `ZSTD_d_forceIgnoreChecksum`.
- Support `ZSTD_d_refMultipleDDicts`.
- Support `ZSTD_generateSequences()`.
- Support `ZSTD_mergeBlockDelimiters()`.
- Support `ZSTD_compressSequences()`.
- Support `ZSTD_threadPool` APIs for managing a thread pool.
- Support `ZSTD_refMultipleDDicts_e`.
- Support `ZSTD_writeSkippableFrame()`.
- Utilize `ZSTD_getDictID_fromCDict()`?
- Utilize `ZSTD_DCtx_getParameter()`.
- Stop relying on private libzstd headers and symbols (namely `pool.h`).

13.1.2 Other Actions Not Blocking Release

- Support for block compression APIs.
- API for ensuring max memory ceiling isn't exceeded.
- Move off nose for testing.

13.2 0.19.0 (released 2022-10-29)

13.2.1 Bug Fixes

- The C backend implementation of `ZstdDecompressionObj.decompress()` could have raised an assertion in cases where the function was called multiple times on an instance. In non-debug builds, calls to this method could have leaked memory.

13.2.2 Changes

- PyPy 3.6 support dropped; Pypy 3.8 and 3.9 support added.
- Anaconda 3.6 support dropped.
- Official support for Python 3.11. This did not require meaningful code changes and previous release(s) likely worked with 3.11 without any changes.
- CFFI's build system now respects distutils's `compiler.preprocessor` if it is set. (#179)
- The internal logic of `ZstdDecompressionObj.decompress()` was refactored. This may have fixed unconfirmed issues where `unused_data` was set prematurely. The new logic will also avoid an extra call to `ZSTD_decompressStream()` in some scenarios, possibly improving performance.
- `ZstdDecompressor.decompress()` how has a `read_across_frames` keyword argument. It defaults to False. True is not yet implemented and will raise an exception if used. The new argument will default to True in a future release and is provided now so callers can start passing `read_across_frames=False` to preserve the existing functionality during a future upgrade.
- `ZstdDecompressor.decompress()` now has an `allow_extra_data` keyword argument to control whether an exception is raised if input contains extra data. It defaults to True, preserving existing behavior of ignoring extra data. It will likely default to False in a future release. Callers desiring the current behavior are encouraged to explicitly pass `allow_extra_data=True` so behavior won't change during a future upgrade.

13.3 0.18.0 (released 2022-06-20)

13.3.1 Changes

- Bundled zstd library upgraded from 1.5.1 to 1.5.2.
- `ZstdDecompressionObj` now has an `unused_data` attribute. It will contain data beyond the fully decoded zstd frame data if said data exists.
- `ZstdDecompressionObj` now has an `unconsumed_tail` attribute. This attribute currently always returns the empty bytes value (`b""`).
- `ZstdDecompressionObj` now has an `eof` attribute returning whether the compressed data has been fully read.

13.4 0.17.0 (released 2021-01-18)

13.4.1 Backwards Compatibility Notes

- `ZstdCompressionWriter` and `ZstdDecompressionWriter` now implement `__iter__()` and `__next__()`. The methods always raise `io.UnsupportedOperation`. The added methods are part of the `io.IOBase` abstract base class / interface and help ensure instances look like other I/O types. (#167, #168)
- The `HASHLOG3_MAX` constant has been removed since it is no longer defined in zstd 1.5.1.

13.4.2 Bug Fixes

- The `ZstdCompressionReader`, `ZstdCompressionWriter`, `ZstdDecompressionReader`, and `ZstdDecompressionWriter` types in the C backend now tracks their `closed` attribute using the proper C type. Before, due to a mismatch between the C struct type and the type declared to Python, Python could read the wrong bits on platforms like s390x and incorrectly report the value of the `closed` attribute to Python. (#105, #164)

13.4.3 Changes

- Bundled zstd library upgraded from 1.5.0 to 1.5.1.
- The C backend now exposes the symbols `ZstdCompressionReader`, `ZstdCompressionWriter`, `ZstdDecompressionReader`, and `ZstdDecompressionWriter`. This should match the behavior of the CFFI backend. (#165)
- `ZstdCompressionWriter` and `ZstdDecompressionWriter` now implement `__iter__` and `__next__`, which always raise `io.UnsupportedOperation`.
- Documentation on thread safety has been updated to note that derived objects like `ZstdCompressionWriter` have the same thread unsafety as the contexts they were derived from. (#166)

13.5 0.16.0 (released 2021-10-16)

13.5.1 Backwards Compatibility Notes

- Support for Python 3.5 has been dropped. Python 3.6 is now the minimum required Python version.

13.5.2 Changes

- Bundled zstd library upgraded from 1.4.8 to 1.5.0.
- `manylinux2014_aarch64` wheels are now being produced for CPython 3.6+. (#145).
- Wheels are now being produced for CPython 3.10.
- Arguments to `ZstdCompressor()` and `ZstdDecompressor()` are now all optional in the C backend and an explicit `None` value is accepted. Before, the C backend wouldn't accept an explicit `None` value (but the CFFI backend would). The new behavior should be consistent between the backends. (#153)

13.6 0.15.2 (released 2021-02-27)

13.6.1 Backwards Compatibility Notes

- `ZstdCompressor.multi_compress_to_buffer()` and `ZstdDecompressor.multi_decompress_to_buffer()` are no longer available when linking against a system zstd library. These experimental features are only available when building against the bundled single file zstd C source file distribution. (#106)

13.6.2 Changes

- `setup.py` now recognizes a `ZSTD_EXTRA_COMPILER_ARGS` environment variable to specify additional compiler arguments to use when compiling the C backend.
- PyPy build and test coverage has been added to CI.
- Added CI jobs for building against external zstd library.
- Wheels supporting macOS ARM/M1 devices are now being produced.
- References to Python 2 have been removed from the in-repo Debian packaging code.
- Significant work has been made on a Rust backend. It is currently feature complete but not yet optimized. We are not yet shipping the backend as part of the distributed wheels until it is more mature.
- The `.pyi` type annotations file has replaced various default argument values with `...`.

13.7 0.15.1 (released 2020-12-31)

13.7.1 Bug Fixes

- `setup.py` no longer attempts to build the C backend on PyPy. (#130)
- `<sys/types.h>` is now included before `<sys/sysctl.h>`. This was the case in releases prior to 0.15.0 and the include order was reversed as part of running `clang-format`. The old/working order has been restored. (#128)
- Include some private zstd C headers so we can build the C extension against a system library. The previous behavior of referencing these headers is restored. That behave is rather questionable and undermines the desire to use the system zstd.

13.8 0.15.0 (released 2020-12-29)

13.8.1 Backwards Compatibility Notes

- Support for Python 2.7 has been dropped. Python 3.5 is now the minimum required Python version. (#109)
- `train_dictionary()` now uses the `fastcover` training mechanism (as opposed to `cover`). Some parameter values that worked with the old mechanism may not work with the new one. e.g. `d` must be `6` or `8` if it is defined.
- `train_dictionary()` now always calls `ZDICT_optimizeTrainFromBuffer_fastCover()` instead of different APIs depending on which arguments were passed.

- The names of various Python modules have been changed. The C extension is now built as `zstandard.backend_c` instead of `zstd`. The CFFI extension module is now built as `zstandard._cffi` instead of `_zstd_cffi`. The CFFI backend is now `zstandard.backend_cffi` instead of `zstandard.cffi`.
- `ZstdDecompressionReader.seekable()` now returns `False` instead of `True` because not all seek operations are supported and some Python code in the wild keys off this value to determine if `seek()` can be called for all scenarios.
- `ZstdDecompressionReader.seek()` now raises `OSError` instead of `ValueError` when the seek cannot be fulfilled. (#107)
- `ZstdDecompressionReader.readline()` and `ZstdDecompressionReader.readlines()` now accept an integer argument. This makes them conform with the IO interface. The methods still raise `io.UnsupportedOperation`.
- `ZstdCompressionReader.__enter__` and `ZstdDecompressionReader.__enter__` now raise `ValueError` if the instance was already closed.
- The deprecated `overlap_size_log` attribute on `ZstdCompressionParameters` instances has been removed. The `overlap_log` attribute should be used instead.
- The deprecated `overlap_size_log` argument to `ZstdCompressionParameters` has been removed. The `overlap_log` argument should be used instead.
- The deprecated `ldm_hash_every_log` attribute on `ZstdCompressionParameters` instances has been removed. The `ldm_hash_rate_log` attribute should be used instead.
- The deprecated `ldm_hash_every_log` argument to `ZstdCompressionParameters` has been removed. The `ldm_hash_rate_log` argument should be used instead.
- The deprecated `CompressionParameters` type alias to `ZstdCompressionParamaters` has been removed. Use `ZstdCompressionParameters`.
- The deprecated aliases `ZstdCompressor.read_from()` and `ZstdDecompressor.read_from()` have been removed. Use the corresponding `read_to_iter()` methods instead.
- The deprecated aliases `ZstdCompressor.write_to()` and `ZstdDecompressor.write_to()` have been removed. Use the corresponding `stream_writer()` methods instead.
- `ZstdCompressor.copy_stream()`, `ZstdCompressorIterator.__next__()`, and `ZstdDecompressor.copy_stream()` now raise the original exception on error calling the source stream's `read()` instead of raising `ZstdError`. This only affects the C backend.
- `ZstdDecompressionObj.flush()` now returns `bytes` instead of `None`. This makes it behave more similarly to `flush()` methods for similar types in the Python standard library. (#78)
- `ZstdCompressionWriter.__exit__()` now always calls `close()`. Previously, `close()` would not be called if the context manager raised an exception. The old behavior was inconsistent with other stream types in this package and with the behavior of Python's standard library IO types. (#86)
- Distribution metadata no longer lists `cffi` as an `install_requires` except when running on PyPy. Instead, `cffi` is listed as an `extras_require`.
- `ZstdCompressor.stream_reader()` and `ZstdDecompressor.stream_reader()` now default to closing the source stream when the instance is itself closed. To change this behavior, pass `closefd=False`. (#76)
- The `CFFI` backend now defines `ZstdCompressor.multi_compress_to_buffer()` and `ZstdDecompressor.multi_decompress_to_buffer()`. However, they raise `NotImplementedError`, as they are not yet implemented.
- The `CFFI` backend now exposes the types `ZstdCompressionChunker`, `ZstdCompressionObj`, `ZstdCompressionReader`, `ZstdCompressionWriter`, `ZstdDecompressionObj`,

ZstdDecompressionReader, and ZstdDecompressionWriter as symbols on the zstandard module.

- The CFFI backend now exposes the types BufferSegment, BufferSegments, BufferWithSegments, and BufferWithSegmentsCollection. However, they are not implemented.
- ZstdCompressionWriter.flush() now calls flush() on the inner stream if such a method exists. However, when close() itself calls self.flush(), flush() is not called on the inner stream.
- ZstdDecompressionWriter.close() no longer calls flush() on the inner stream. However, ZstdDecompressionWriter.flush() still calls flush() on the inner stream.
- ZstdCompressor.stream_writer() and ZstdDecompressor.stream_writer() now have their write_return_read argument default to True. This brings the behavior of write() in compliance with the io.RawIOBase interface by default. The argument may be removed in a future release.
- ZstdCompressionParameters no longer exposes a compression_strategy property. Its constructor no longer accepts a compression_strategy argument. Use the strategy property/argument instead.

13.8.2 Bug Fixes

- Fix a memory leak in stream_reader decompressor when reader is closed before reading everything. (Patch by Pierre Fersing.)
- The C backend now properly checks for errors after calling IO methods on inner streams in various methods. ZstdCompressionWriter.write() now catches exceptions when calling the inner stream's write(). ZstdCompressionWriter.flush() on inner stream's write(). ZstdCompressor.copy_stream() on dest stream's write(). ZstdDecompressionWriter.write() on inner stream's write(). ZstdDecompressor.copy_stream() on dest stream's write(). (#102)

13.8.3 Changes

- Bundled zstandard library upgraded from 1.4.5 to 1.4.8.
- The bundled zstandard library is now using the single C source file distribution. The 2 main header files are still present, as these are needed by CFFI to generate the CFFI bindings.
- PyBuffer instances are no longer checked to be C contiguous and have a single dimension. The former was redundant with what PyArg_ParseTuple() already did and the latter is not necessary in practice because very few extension modules create buffers with more than 1 dimension. (#124)
- Added Python typing stub file for the zstandard module. (#120)
- The make_cffi.py script should now respect the CC environment variable for locating the compiler. (#103)
- CI now properly uses the cffi backend when running all tests.
- train_dictionary() has been rewritten to use the fastcover APIs and to consistently call ZDICT_optimizeTrainFromBuffer_fastCover() instead of different C APIs depending on what arguments were passed. The function also now accepts arguments f, split_point, and accel, which are parameters unique to fastcover.
- CI now tests and builds wheels for Python 3.9.
- zstd.c file renamed to c-ext/backend_c.c.
- All built/installed Python modules are now in the zstandard package. Previously, there were modules in other packages. (#115)

- C source code is now automatically formatted with `clang-format`.
- `ZstdCompressor.stream_writer()`, `ZstdCompressor.stream_reader()`, `ZstdDecompressor.stream_writer()`, and `ZstdDecompressor.stream_reader()` now accept a `closefd` argument to control whether the underlying stream should be closed when the `ZstdCompressionWriter`, `ZstdCompressReader`, `ZstdDecompressionWriter`, or `ZstdDecompressionReader` is closed. (#76)
- There is now a `zstandard.open()` function for returning a file object with zstd (de)compression. (#64)
- The `zstandard` module now exposes a `backend_features` attribute containing a set of strings denoting optional features present in that backend. This can be used to sniff feature support by performing a string lookup instead of sniffing for API presence or behavior.
- Python docstrings have been moved from the C backend to the CFFI backend. Sphinx docs have been updated to generate API documentation via the CFFI backend. Documentation for Python APIs is now fully defined via Python docstrings instead of spread across Sphinx ReST files and source code.
- `ZstdCompressionParameters` now exposes a `strategy` property.
- There are now `compress()` and `decompress()` convenience functions on the `zstandard` module. These are simply wrappers around the corresponding APIs on `ZstdCompressor` and `ZstdDecompressor`.

13.9 0.14.1 (released 2020-12-05)

13.9.1 Changes

- Python 3.9 wheels are now provided.

13.10 0.14.0 (released 2020-06-13)

13.10.1 Backwards Compatibility Notes

- This will likely be the final version supporting Python 2.7. Future releases will likely only work on Python 3.5+. See #109 for more context.
- There is a significant possibility that future versions will use Rust - instead of C - for compiled code. See #110 for more context.

13.10.2 Bug Fixes

- Some internal fields of C structs are now explicitly initialized. (Possible fix for #105.)
- The `make_cffi.py` script used to build the CFFI bindings now calls `distutils.sysconfig.customize_compiler()` so compiler customizations (such as honoring the `CC` environment variable) are performed. Patch by @Arfrever. (#103)
- The `make_cffi.py` script now sets `LC_ALL=C` when invoking the preprocessor in an attempt to normalize output to ASCII. (#95)

13.10.3 Changes

- Bundled zstandard library upgraded from 1.4.4 to 1.4.5.
- `setup.py` is now executable.
- Python code reformatted with black using 80 character line lengths.

13.11 0.13.0 (released 2019-12-28)

13.11.1 Changes

- `pytest-xdist` `pytest` extension is now installed so tests can be run in parallel.
- CI now builds `manylinux2010` and `manylinux2014` binary wheels instead of a mix of `manylinux2010` and `manylinux1`.
- Official support for Python 3.8 has been added.
- Bundled zstandard library upgraded from 1.4.3 to 1.4.4.
- Python code has been reformatted with black.

13.12 0.12.0 (released 2019-09-15)

13.12.1 Backwards Compatibility Notes

- Support for Python 3.4 has been dropped since Python 3.4 is no longer a supported Python version upstream. (But it will likely continue to work until Python 2.7 support is dropped and we port to Python 3.5+ APIs.)

13.12.2 Bug Fixes

- Fix `ZstdDecompressor.__init__` on 64-bit big-endian systems (#91).
- Fix memory leak in `ZstdDecompressionReader.seek()` (#82).

13.12.3 Changes

- CI transitioned to Azure Pipelines (from AppVeyor and Travis CI).
- Switched to `pytest` for running tests (from `nose`).
- Bundled zstandard library upgraded from 1.3.8 to 1.4.3.

13.13 0.11.1 (released 2019-05-14)

- Fix memory leak in `ZstdDecompressionReader.seek()` (#82).

13.14 0.11.0 (released 2019-02-24)

13.14.1 Backwards Compatibility Notes

- `ZstdDecompressor.read()` now allows reading sizes of `-1` or `0` and defaults to `-1`, per the documented behavior of `io.RawIOBase.read()`. Previously, we required an argument that was a positive value.
- The `readline()`, `readlines()`, `__iter__`, and `__next__` methods of `ZstdDecompressionReader()` now raise `io.UnsupportedOperation` instead of `NotImplementedError`.
- `ZstdDecompressor.stream_reader()` now accepts a `read_across_frames` argument. The default value will likely be changed in a future release and consumers are advised to pass the argument to avoid unwanted change of behavior in the future.
- `setup.py` now always disables the CFFI backend if the installed CFFI package does not meet the minimum version requirements. Before, it was possible for the CFFI backend to be generated and a run-time error to occur.
- In the CFFI backend, `CompressionReader` and `DecompressionReader` were renamed to `ZstdCompressionReader` and `ZstdDecompressionReader`, respectively so naming is identical to the C extension. This should have no meaningful end-user impact, as instances aren't meant to be constructed directly.
- `ZstdDecompressor.stream_writer()` now accepts a `write_return_read` argument to control whether `write()` returns the number of bytes read from the source / written to the decompressor. It defaults to off, which preserves the existing behavior of returning the number of bytes emitted from the decompressor. The default will change in a future release so behavior aligns with the specified behavior of `io.RawIOBase`.
- `ZstdDecompressionWriter.__exit__` now calls `self.close()`. This will result in that stream plus the underlying stream being closed as well. If this behavior is not desirable, do not use instances as context managers.
- `ZstdCompressor.stream_writer()` now accepts a `write_return_read` argument to control whether `write()` returns the number of bytes read from the source / written to the compressor. It defaults to off, which preserves the existing behavior of returning the number of bytes emitted from the compressor. The default will change in a future release so behavior aligns with the specified behavior of `io.RawIOBase`.
- `ZstdCompressionWriter.__exit__` now calls `self.close()`. This will result in that stream plus any underlying stream being closed as well. If this behavior is not desirable, do not use instances as context managers.
- `ZstdDecompressionWriter` no longer requires being used as a context manager (#57).
- `ZstdCompressionWriter` no longer requires being used as a context manager (#57).
- The `overlap_size_log` attribute on `CompressionParameters` instances has been deprecated and will be removed in a future release. The `overlap_log` attribute should be used instead.
- The `overlap_size_log` argument to `CompressionParameters` has been deprecated and will be removed in a future release. The `overlap_log` argument should be used instead.
- The `ldm_hash_every_log` attribute on `CompressionParameters` instances has been deprecated and will be removed in a future release. The `ldm_hash_rate_log` attribute should be used instead.
- The `ldm_hash_every_log` argument to `CompressionParameters` has been deprecated and will be removed in a future release. The `ldm_hash_rate_log` argument should be used instead.
- The `compression_strategy` argument to `CompressionParameters` has been deprecated and will be removed in a future release. The `strategy` argument should be used instead.

- The SEARCHLENGTH_MIN and SEARCHLENGTH_MAX constants are deprecated and will be removed in a future release. Use MINMATCH_MIN and MINMATCH_MAX instead.
- The zstd_cffi module has been renamed to zstandard.cffi. As had been documented in the README file since the 0.9.0 release, the module should not be imported directly at its new location. Instead, import zstandard to cause an appropriate backend module to be loaded automatically.

13.14.2 Bug Fixes

- CFFI backend could encounter a failure when sending an empty chunk into ZstdDecompressionObj.decompress(). The issue has been fixed.
- CFFI backend could encounter an error when calling ZstdDecompressionReader.read() if there was data remaining in an internal buffer. The issue has been fixed. (#71)

13.14.3 Changes

- ZstdDecompressionObj.decompress() now properly handles empty inputs in the CFFI backend.
- ZstdCompressionReader now implements read1() and readinto1(). These are part of the io.BufferedIOBase interface.
- ZstdCompressionReader has gained a readinto(b) method for reading compressed output into an existing buffer.
- ZstdCompressionReader.read() now defaults to size=-1 and accepts read sizes of -1 and 0. The new behavior aligns with the documented behavior of io.RawIOBase.
- ZstdCompressionReader now implements readall(). Previously, this method raised NotImplementedError.
- ZstdDecompressionReader now implements read1() and readinto1(). These are part of the io.BufferedIOBase interface.
- ZstdDecompressionReader.read() now defaults to size=-1 and accepts read sizes of -1 and 0. The new behavior aligns with the documented behavior of io.RawIOBase.
- ZstdDecompressionReader() now implements readall(). Previously, this method raised NotImplementedError.
- The readline(), readlines(), __iter__, and __next__ methods of ZstdDecompressionReader() now raise io.UnsupportedOperation instead of NotImplementedError. This reflects a decision to never implement text-based I/O on (de)compressors and keep the low-level API operating in the binary domain. (#13)
- README.rst now documented how to achieve linewise iteration using an io.TextIOWrapper with a ZstdDecompressionReader.
- ZstdDecompressionReader has gained a readinto(b) method for reading decompressed output into an existing buffer. This allows chaining to an io.TextIOWrapper on Python 3 without using an io.BufferedReader.
- ZstdDecompressor.stream_reader() now accepts a read_across_frames argument to control behavior when the input data has multiple zstd *frames*. When False (the default for backwards compatibility), a read() will stop when the end of a zstd *frame* is encountered. When True, read() can potentially return data spanning multiple zstd *frames*. The default will likely be changed to True in a future release.

- `setup.py` now performs CFFI version sniffing and disables the CFFI backend if CFFI is too old. Previously, we only used `install_requires` to enforce the CFFI version and not all build modes would properly enforce the minimum CFFI version. (#69)
- CFFI's `ZstdDecompressionReader.read()` now properly handles data remaining in any internal buffer. Before, repeated `read()` could result in *random* errors. (#71)
- Upgraded various Python packages in CI environment.
- Upgrade to hypothesis 4.5.11.
- In the CFFI backend, `CompressionReader` and `DecompressionReader` were renamed to `ZstdCompressionReader` and `ZstdDecompressionReader`, respectively.
- `ZstdDecompressor.stream_writer()` now accepts a `write_return_read` argument to control whether `write()` returns the number of bytes read from the source. It defaults to `False` to preserve backwards compatibility.
- `ZstdDecompressor.stream_writer()` now implements the `io.RawIOBase` interface and behaves as a proper stream object.
- `ZstdCompressor.stream_writer()` now accepts a `write_return_read` argument to control whether `write()` returns the number of bytes read from the source. It defaults to `False` to preserve backwards compatibility.
- `ZstdCompressionWriter` now implements the `io.RawIOBase` interface and behaves as a proper stream object. `close()` will now close the stream and the underlying stream (if possible). `__exit__` will now call `close()`. Methods like `writable()` and `fileno()` are implemented.
- `ZstdDecompressionWriter` no longer must be used as a context manager.
- `ZstdCompressionWriter` no longer must be used as a context manager. When not using as a context manager, it is important to call `flush(FRAME_FRAME)` or the compression stream won't be properly terminated and decoders may complain about malformed input.
- `ZstdCompressionWriter.flush()` (what is returned from `ZstdCompressor.stream_writer()`) now accepts an argument controlling the flush behavior. Its value can be one of the new constants `FLUSH_BLOCK` or `FLUSH_FRAME`.
- `ZstdDecompressionObj` instances now have a `flush([length=None])` method. This provides parity with standard library equivalent types. (#65)
- `CompressionParameters` no longer redundantly store individual compression parameters on each instance. Instead, compression parameters are stored inside the underlying `ZSTD_CCtx_params` instance. Attributes for obtaining parameters are now properties rather than instance variables.
- Exposed the `STRATEGY_BTULTRA2` constant.
- `CompressionParameters` instances now expose an `overlap_log` attribute. This behaves identically to the `overlap_size_log` attribute.
- `CompressionParameters()` now accepts an `overlap_log` argument that behaves identically to the `overlap_size_log` argument. An error will be raised if both arguments are specified.
- `CompressionParameters` instances now expose an `ldm_hash_rate_log` attribute. This behaves identically to the `ldm_hash_every_log` attribute.
- `CompressionParameters()` now accepts a `ldm_hash_rate_log` argument that behaves identically to the `ldm_hash_every_log` argument. An error will be raised if both arguments are specified.
- `CompressionParameters()` now accepts a `strategy` argument that behaves identically to the `compression_strategy` argument. An error will be raised if both arguments are specified.

- The `MINMATCH_MIN` and `MINMATCH_MAX` constants were added. They are semantically equivalent to the old `SEARCHLENGTH_MIN` and `SEARCHLENGTH_MAX` constants.
- Bundled zstandard library upgraded from 1.3.7 to 1.3.8.
- `setup.py` denotes support for Python 3.7 (Python 3.7 was supported and tested in the 0.10 release).
- `zstd_cffi` module has been renamed to `zstandard.cffi`.
- `ZstdCompressor.stream_writer()` now reuses a buffer in order to avoid allocating a new buffer for every operation. This should result in faster performance in cases where `write()` or `flush()` are being called frequently. (#62)
- Bundled zstandard library upgraded from 1.3.6 to 1.3.7.

13.15 0.10.2 (released 2018-11-03)

13.15.1 Bug Fixes

- `zstd_cffi.py` added to `setup.py` (#60).

13.15.2 Changes

- Change some integer casts to avoid `ssize_t` (#61).

13.16 0.10.1 (released 2018-10-08)

13.16.1 Backwards Compatibility Notes

- `ZstdCompressor.stream_reader().closed` is now a property instead of a method (#58).
- `ZstdDecompressor.stream_reader().closed` is now a property instead of a method (#58).

13.16.2 Changes

- Stop attempting to package Python 3.6 for Miniconda. The latest version of Miniconda is using Python 3.7. The Python 3.6 Miniconda packages were a lie since this were built against Python 3.7.
- `ZstdCompressor.stream_reader()`'s and `ZstdDecompressor.stream_reader()`'s `closed` attribute is now a read-only property instead of a method. This now properly matches the `IOBase` API and allows instances to be used in more places that accept `IOBase` instances.

13.17 0.10.0 (released 2018-10-08)

13.17.1 Backwards Compatibility Notes

- `ZstdDecompressor.stream_reader().read()` now consistently requires an argument in both the C and CFFI backends. Before, the CFFI implementation would assume a default value of `-1`, which was later rejected.

- The `compress_literals` argument and attribute has been removed from `zstd.ZstdCompressionParameters` because it was removed by the zstd 1.3.5 API.
- `ZSTD_CCtx_setParametersUsingCCtxParams()` is no longer called on every operation performed against `ZstdCompressor` instances. The reason for this change is that the zstd 1.3.5 API no longer allows this without calling `ZSTD_CCtx_resetParameters()` first. But if we called `ZSTD_CCtx_resetParameters()` on every operation, we'd have to redo potentially expensive setup when using dictionaries. We now call `ZSTD_CCtx_reset()` on every operation and don't attempt to change compression parameters.
- Objects returned by `ZstdCompressor.stream_reader()` no longer need to be used as a context manager. The context manager interface still exists and its behavior is unchanged.
- Objects returned by `ZstdDecompressor.stream_reader()` no longer need to be used as a context manager. The context manager interface still exists and its behavior is unchanged.

13.17.2 Bug Fixes

- `ZstdDecompressor.decompressobj().decompress()` should now return all data from internal buffers in more scenarios. Before, it was possible for data to remain in internal buffers. This data would be emitted on a subsequent call to `decompress()`. The overall output stream would still be valid. But if callers were expecting input data to exactly map to output data (say the producer had used `flush(COMPRESSOBJ_FLUSH_BLOCK)` and was attempting to map input chunks to output chunks), then the previous behavior would be wrong. The new behavior is such that output from `flush(COMPRESSOBJ_FLUSH_BLOCK)` fed into `decompressobj().decompress()` should produce all available compressed input.
- `ZstdDecompressor.stream_reader().read()` should no longer segfault after a previous context manager resulted in error (#56).
- `ZstdCompressor.compressobj().flush(COMPRESSOBJ_FLUSH_BLOCK)` now returns all data necessary to flush a block. Before, it was possible for the `flush()` to not emit all data necessary to fully represent a block. This would mean decompressors wouldn't be able to decompress all data that had been fed into the compressor and `flush()`'ed. (#55).

13.17.3 New Features

- New module constants `BLOCKSIZELOG_MAX`, `BLOCKSIZE_MAX`, `TARGETLENGTH_MAX` that expose constants from libzstd.
- New `ZstdCompressor.chunker()` API for manually feeding data into a compressor and emitting chunks of a fixed size. Like `compressobj()`, the API doesn't impose restrictions on the input or output types for the data streams. Unlike `compressobj()`, it ensures output chunks are of a fixed size. This makes this API useful when the compressed output is being fed into an I/O layer, where uniform write sizes are useful.
- `ZstdCompressor.stream_reader()` no longer needs to be used as a context manager (#34).
- `ZstdDecompressor.stream_reader()` no longer needs to be used as a context manager (#34).
- Bundled zstandard library upgraded from 1.3.4 to 1.3.6.

13.17.4 Changes

- Added `zstd_cffi.py` and `NEWS.rst` to `MANIFEST.in`.
- `zstandard.__version__` is now defined (#50).

- Upgrade pip, setuptools, wheel, and cibuildwheel packages to latest versions.
- Upgrade various packages used in CI to latest versions. Notably tox (in order to support Python 3.7).
- Use relative paths in setup.py to appease Python 3.7 (#51).
- Added CI for Python 3.7.

13.18 0.9.1 (released 2018-06-04)

- Debian packaging support.
- Fix typo in setup.py (#44).
- Support building with mingw compiler (#46).

13.19 0.9.0 (released 2018-04-08)

13.19.1 Backwards Compatibility Notes

- CFFI 1.11 or newer is now required (previous requirement was 1.8).
- The primary module is now `zstandard`. Please change imports of `zstd` and `zstd_cffi` to `import zstandard`. See the README for more. Support for importing the old names will be dropped in the next release.
- `ZstdCompressor.read_from()` and `ZstdDecompressor.read_from()` have been renamed to `read_to_iter()`. `read_from()` is aliased to the new name and will be deleted in a future release.
- Support for Python 2.6 has been removed.
- Support for Python 3.3 has been removed.
- The `selectivity` argument to `train_dictionary()` has been removed, as the feature disappeared from zstd 1.3.
- Support for legacy dictionaries has been removed. Cover dictionaries are now the default. `train_cover_dictionary()` has effectively been renamed to `train_dictionary()`.
- The `allow_empty` argument from `ZstdCompressor.compress()` has been deleted and the method now allows empty inputs to be compressed by default.
- `estimate_compression_context_size()` has been removed. Use `CompressionParameters.estimated_compression_context_size()` instead.
- `get_compression_parameters()` has been removed. Use `CompressionParameters.from_level()` instead.
- The arguments to `CompressionParameters.__init__()` have changed. If you were using positional arguments before, the positions now map to different arguments. It is recommended to use keyword arguments to construct `CompressionParameters` instances.
- `TARGETLENGTH_MAX` constant has been removed (it disappeared from zstandard 1.3.4).
- `ZstdCompressor.write_to()` and `ZstdDecompressor.write_to()` have been renamed to `ZstdCompressor.stream_writer()` and `ZstdDecompressor.stream_writer()`, respectively. The old names are still aliased, but will be removed in the next major release.

- Content sizes are written into frame headers by default (`ZstdCompressor(write_content_size=True)` is now the default).
- `CompressionParameters` has been renamed to `ZstdCompressionParameters` for consistency with other types. The old name is an alias and will be removed in the next major release.

13.19.2 Bug Fixes

- Fixed memory leak in `ZstdCompressor.copy_stream()` (#40) (from 0.8.2).
- Fixed memory leak in `ZstdDecompressor.copy_stream()` (#35) (from 0.8.2).
- Fixed memory leak of `ZSTD_DDict` instances in CFFI's `ZstdDecompressor`.

13.19.3 New Features

- Bundled zstandard library upgraded from 1.1.3 to 1.3.4. This delivers various bug fixes and performance improvements. It also gives us access to newer features.
- Support for negative compression levels.
- Support for *long distance matching* (facilitates compression ratios that approach LZMA).
- Supporting for reading empty zstandard frames (with an embedded content size of 0).
- Support for writing and partial support for reading zstandard frames without a magic header.
- New `stream_reader()` API that exposes the `io.RawIOBase` interface (allows you to `.read()` from a file-like object).
- Several minor features, bug fixes, and performance enhancements.
- Wheels for Linux and macOS are now provided with releases.

13.19.4 Changes

- Functions accepting bytes data now use the buffer protocol and can accept more types (like `memoryview` and `bytearray`) (#26).
- Add #includes so compilation on OS X and BSDs works (#20).
- New `ZstdDecompressor.stream_reader()` API to obtain a read-only i/o stream of decompressed data for a source.
- New `ZstdCompressor.stream_reader()` API to obtain a read-only i/o stream of compressed data for a source.
- Renamed `ZstdDecompressor.read_from()` to `ZstdDecompressor.read_to_iter()`. The old name is still available.
- Renamed `ZstdCompressor.read_from()` to `ZstdCompressor.read_to_iter()`. `read_from()` is still available at its old location.
- Introduce the `zstandard` module to import and re-export the C or CFFI *backend* as appropriate. Behavior can be controlled via the `PYTHON_ZSTANDARD_IMPORT_POLICY` environment variable. See README for usage info.
- Vendored version of zstd upgraded to 1.3.4.
- Added module constants `CONTENTSIZE_UNKNOWN` and `CONTENTSIZE_ERROR`.

- Add `STRATEGY_BTULTRA` compression strategy constant.
- Switch from deprecated `ZSTD_getDecompressedSize()` to `ZSTD_getFrameContentSize()` replacement.
- `ZstdCompressor.compress()` can now compress empty inputs without requiring special handling.
- `ZstdCompressor` and `ZstdDecompressor` now have a `memory_size()` method for determining the current memory utilization of the underlying zstd primitive.
- `train_dictionary()` has new arguments and functionality for trying multiple variations of COVER parameters and selecting the best one.
- Added module constants `LDM_MINMATCH_MIN`, `LDM_MINMATCH_MAX`, and `LDM_BUCKETSIZELOG_MAX`.
- Converted all consumers to the zstandard *new advanced API*, which uses `ZSTD_compress_generic()`
- `CompressionParameters.__init__` now accepts several more arguments, including support for *long distance matching*.
- `ZstdCompressionDict.__init__` now accepts a `dict_type` argument that controls how the dictionary should be interpreted. This can be used to force the use of *content-only* dictionaries or to require the presence of the dictionary magic header.
- `ZstdCompressionDict.precompute_compress()` can be used to precompute the compression dictionary so it can efficiently be used with multiple `ZstdCompressor` instances.
- Digested dictionaries are now stored in `ZstdCompressionDict` instances, created automatically on first use, and automatically reused by all `ZstdDecompressor` instances bound to that dictionary.
- All meaningful functions now accept keyword arguments.
- `ZstdDecompressor.decompressobj()` now accepts a `write_size` argument to control how much work to perform on every decompressor invocation.
- `ZstdCompressor.write_to()` now exposes a `tell()`, which exposes the total number of bytes written so far.
- `ZstdDecompressor.stream_reader()` now supports `seek()` when moving forward in the stream.
- Removed `TARGETLENGTH_MAX` constant.
- Added `frame_header_size(data)` function.
- Added `frame_content_size(data)` function.
- Consumers of `ZSTD_decompress*` have been switched to the new *advanced decompression* API.
- `ZstdCompressor` and `ZstdCompressionParams` can now be constructed with negative compression levels.
- `ZstdDecompressor` now accepts a `max_window_size` argument to limit the amount of memory required for decompression operations.
- `FORMAT_ZSTD1` and `FORMAT_ZSTD1_MAGICLESS` constants to be used with the `format` compression parameter to control whether the frame magic header is written.
- `ZstdDecompressor` now accepts a `format` argument to control the expected frame format.
- `ZstdCompressor` now has a `frame_progression()` method to return information about the current compression operation.
- Error messages in CFFI no longer have `b''` literals.
- Compiler warnings and underlying overflow issues on 32-bit platforms have been fixed.

- Builds in CI now build with compiler warnings as errors. This should hopefully fix new compiler warnings from being introduced.
- Make `ZstdCompressor(write_content_size=True)` and `CompressionParameters(write_content_size` the default.
- `CompressionParameters` has been renamed to `ZstdCompressionParameters`.

13.19.5 0.8.2 (released 2018-02-22)

- Fixed memory leak in `ZstdCompressor.copy_stream()` (#40).
- Fixed memory leak in `ZstdDecompressor.copy_stream()` (#35).

13.19.6 0.8.1 (released 2017-04-08)

- Add #includes so compilation on OS X and BSDs works (#20).

13.20 0.8.0 (released 2017-03-08)

- CompressionParameters now has a estimated_compression_context_size() method. zstd.estimate_compression_context_size() is now deprecated and slated for removal.
- Implemented a lot of fuzzing tests.
- CompressionParameters instances now perform extra validation by calling ZSTD_checkCParams() at construction time.
- multi_compress_to_buffer() API for compressing multiple inputs as a single operation, as efficiently as possible.
- ZSTD_CStream instances are now used across multiple operations on ZstdCompressor instances, resulting in much better performance for APIs that do streaming.
- ZSTD_DStream instances are now used across multiple operations on ZstdDecompressor instances, resulting in much better performance for APIs that do streaming.
- train_dictionary() now releases the GIL.
- Support for training dictionaries using the COVER algorithm.
- multi_decompress_to_buffer() API for decompressing multiple frames as a single operation, as efficiently as possible.
- Support for multi-threaded compression.
- Disable deprecation warnings when compiling CFFI module.
- Fixed memory leak in train_dictionary().
- Removed DictParameters type.
- train_dictionary() now accepts keyword arguments instead of a DictParameters instance to control dictionary generation.

13.21 0.7.0 (released 2017-02-07)

- Added zstd.get_frame_parameters() to obtain info about a zstd frame.
- Added ZstdDecompressor.decompress_content_dict_chain() for efficient decompression of *content-only dictionary chains*.
- CFFI module fully implemented; all tests run against both C extension and CFFI implementation.
- Vendored version of zstd updated to 1.1.3.
- Use ZstdDecompressor.decompress() now uses ZSTD_createDDict_byReference() to avoid extra memory allocation of dict data.
- Add function names to error messages (by using ":name" in PyArg_Parse* functions).
- Reuse decompression context across operations. Previously, we created a new ZSTD_DCtx for each decompress(). This was measured to slow down decompression by 40-200MB/s. The API guarantees say ZstdDecompressor is not thread safe. So we reuse the ZSTD_DCtx across operations and make things faster in the process.
- ZstdCompressor.write_to()'s compress() and flush() methods now return number of bytes written.
- ZstdDecompressor.write_to()'s write() method now returns the number of bytes written to the underlying output object.
- CompressionParameters instances now expose their values as attributes.
- CompressionParameters instances no longer are subscriptable nor behave as tuples (backwards incompatible). Use attributes to obtain values.
- DictParameters instances now expose their values as attributes.

13.22 0.6.0 (released 2017-01-14)

- Support for legacy zstd protocols (build time opt in feature).
- Automation improvements to test against Python 3.6, latest versions of Tox, more deterministic AppVeyor behavior.
- CFFI "parser" improved to use a compiler preprocessor instead of rewriting source code manually.
- Vendored version of zstd updated to 1.1.2.
- Documentation improvements.
- Introduce a bench.py script for performing (crude) benchmarks.
- ZSTD_CCtx instances are now reused across multiple compress() operations.
- ZstdCompressor.write_to() now has a flush() method.
- ZstdCompressor.compressobj()'s flush() method now accepts an argument to flush a block (as opposed to ending the stream).
- Disallow compress(b'') when writing content sizes by default (issue #11).

13.23 0.5.2 (released 2016-11-12)

- more packaging fixes for source distribution

13.24 0.5.1 (released 2016-11-12)

- setup_zstd.py is included in the source distribution

13.25 0.5.0 (released 2016-11-10)

- Vendored version of zstd updated to 1.1.1.
- Continuous integration for Python 3.6 and 3.7
- Continuous integration for Conda
- Added compression and decompression APIs providing similar interfaces to the standard library `zlib` and `bz2` modules. This allows coding to a common interface.
- `zstd.__version__` is now defined.
- `read_from()` on various APIs now accepts objects implementing the buffer protocol.
- `read_from()` has gained a `skip_bytes` argument. This allows callers to pass in an existing buffer with a header without having to create a slice or a new object.
- Implemented `ZstdCompressionDict.as_bytes()`.
- Python's memory allocator is now used instead of `malloc()`.
- Low-level zstd data structures are reused in more instances, cutting down on overhead for certain operations.
- `distutils` boilerplate for obtaining an `Extension` instance has now been refactored into a standalone `setup_zstd.py` file. This allows other projects with `setup.py` files to reuse the `distutils` code for this project without copying code.
- The monolithic `zstd.c` file has been split into a header file defining types and separate `.c` source files for the implementation.

13.26 Older History

2016-08-31 - Zstandard 1.0.0 is released and Gregory starts hacking on a Python extension for use by the Mercurial project. A very hacky prototype is sent to the mercurial-devel list for RFC.

2016-09-03 - Most functionality from Zstandard C API implemented. Source code published on https://github.com/indygreg/python-zstandard. Travis-CI automation configured. 0.0.1 release on PyPI.

2016-09-05 - After the API was rounded out a bit and support for Python 2.6 and 2.7 was added, version 0.1 was released to PyPI.

2016-09-05 - After the compressor and decompressor APIs were changed, 0.2 was released to PyPI.

2016-09-10 - 0.3 is released with a bunch of new features. ZstdCompressor now accepts arguments controlling frame parameters. The source size can now be declared when performing streaming compression. ZstdDecompressor.decompress() is implemented. Compression dictionaries are now cached when using the simple compression and decompression APIs. Memory size APIs added. ZstdCompressor.read_from() and ZstdDecompressor.read_from() have been implemented. This rounds out the major compression/decompression APIs planned by the author.

2016-10-02 - 0.3.3 is released with a bug fix for read_from not fully decoding a zstd frame (issue #2).

2016-10-02 - 0.4.0 is released with zstd 1.1.0, support for custom read and write buffer sizes, and a few bug fixes involving failure to read/write all data when buffer sizes were too small to hold remaining data.

2016-11-10 - 0.5.0 is released with zstd 1.1.1 and other enhancements.

Index

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Z