`!compression.zstd` --- Compression compatible with the Zstandard format

- Exceptions
- Reading and writing compressed files
- Compressing and decompressing data in memory
- Zstandard dictionaries
- Advanced parameter control
- Miscellaneous
- Examples

compression.zstd

3.14

Source code: `Lib/compression/zstd/__init__.py`

This module provides classes and functions for compressing and decompressing data using the Zstandard (or *zstd*) compression algorithm. The zstd manual describes Zstandard as "a fast lossless compression algorithm, targeting real-time compression scenarios at zlib-level and better compression ratios." Also included is a file interface that supports reading and writing the contents of `.zst` files created by the `zstd` utility, as well as raw zstd compressed streams.

The `!compression.zstd` module contains:

- The `.open` function and `ZstdFile` class for reading and writing compressed files.
- The `ZstdCompressor` and `ZstdDecompressor` classes for incremental (de)compression.
- The `compress` and `decompress` functions for one-shot (de)compression.
- The `train_dict` and `finalize_dict` functions and the `ZstdDict` class to train and manage Zstandard dictionaries.

- The `CompressionParameter`, `DecompressionParameter`, and `Strategy` classes for setting advanced (de)compression parameters.

This is an `optional module`. If it is missing from your copy of CPython, look for documentation from your distributor (that is, whoever provided Python to you). If you are the distributor, see `optional-module-requirements`.

Exceptions

ZstdError

This exception is raised when an error occurs during compression or decompression, or while initializing the (de)compressor state.

Reading and writing compressed files

open(file, /, mode='rb', *, level=None, options=None, zstd_dict=None, encoding=None, errors=None, newline=None)

Open a Zstandard-compressed file in binary or text mode, returning a `file object`.

The *file* argument can be either a file name (given as a `str`, `bytes` or `path-like <path-like object>` object), in which case the named file is opened, or it can be an existing file object to read from or write to.

The mode argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'ab'` for appending, or `'xb'` for exclusive creation. These can equivalently be given as `'r'`, `'w'`, `'a'`, and `'x'` respectively. You may also open in text mode with `'rt'`, `'wt'`, `'at'`, and `'xt'` respectively.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a `!TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced compression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be non-None. The *zstd_dict* argument is a `ZstdDict` instance to be used during

compression.

In binary mode, this function is equivalent to the `ZstdFile` constructor: `ZstdFile(file, mode, ...)`. In this case, the *encoding*, *errors*, and *newline* parameters must not be provided.

In text mode, a `ZstdFile` object is created, and wrapped in an `io.TextIOWrapper` instance with the specified encoding, error handling behavior, and line endings.

Open a Zstandard-compressed file in binary mode.

A `ZstdFile` can wrap an already-open `file object`, or operate directly on a named file. The *file* argument specifies either the file object to wrap, or the name of the file to open (as a `str`, `bytes` or `path-like <path-like object>` object). If wrapping an existing file object, the wrapped file will not be closed when the `ZstdFile` is closed.

The *mode* argument can be either `'rb'` for reading (default), `'wb'` for overwriting, `'xb'` for exclusive creation, or `'ab'` for appending. These can equivalently be given as `'r'`, `'w'`, `'x'` and `'a'` respectively.

If *file* is a file object (rather than an actual file name), a mode of `'w'` does not truncate the file, and is instead equivalent to `'a'`.

When reading, the *options* argument can be a dictionary providing advanced decompression parameters; see `DecompressionParameter` for detailed information about supported parameters. The *zstd_dict* argument is a `ZstdDict` instance to be used during decompression. When reading, if the *level* argument is not None, a `!TypeError` will be raised.

When writing, the *options* argument can be a dictionary providing advanced compression parameters; see `CompressionParameter` for detailed information about supported parameters. The *level* argument is the compression level to use when writing compressed data. Only one of *level* or *options* may be passed. The *zstd_dict* argument is a `ZstdDict` instance to be used during compression.

`!ZstdFile` supports all the members specified by `io.BufferedIOBase`, except for `~io.BufferedIOBase.detach` and `~io.IOBase.truncate`. Iteration and the `with` statement are supported.

The following method and attributes are also provided:

peek(size=-1)

Return buffered data without advancing the file position. At least one byte of data will be returned, unless EOF has been reached. The exact number of bytes returned is unspecified (the *size* argument is ignored).

> While calling `peek` does not change the file position of the `ZstdFile`, it may change the position of the underlying file object (for example, if the `ZstdFile` was constructed by passing a file object for *file*).

mode

`'rb'` for reading and `'wb'` for writing.

name

The name of the Zstandard file. Equivalent to the `~io.FileIO.name` attribute of the underlying `file object`.

Compressing and decompressing data in memory

compress(data, level=None, options=None, zstd_dict=None)

Compress *data* (a `bytes-like object`), returning the compressed data as a `bytes` object.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `~CompressionParameter.bounds` on `~CompressionParameter.compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `!CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict` can be used to generate a Zstandard dictionary.

decompress(data, zstd_dict=None, options=None)

Decompress *data* (a `bytes-like object`), returning the uncompressed data as a `bytes` object.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

If *data* is the concatenation of multiple distinct compressed frames, decompress all of these frames, and return the concatenation of the results.

Create a compressor object, which can be used to compress data incrementally.

For a more convenient way of compressing a single chunk of data, see the module-level function `compress`.

The *level* argument is an integer controlling the level of compression. *level* is an alternative to setting `CompressionParameter.compression_level` in *options*. Use `~CompressionParameter.bounds` on `~CompressionParameter.compression_level` to get the values that can be passed for *level*. If advanced compression options are needed, the *level* argument must be omitted and in the *options* dictionary the `!CompressionParameter.compression_level` parameter should be set.

The *options* argument is a Python dictionary containing advanced compression parameters. The valid keys and values for compression parameters are documented as part of the `CompressionParameter` documentation.

The *zstd_dict* argument is an optional instance of `ZstdDict` containing trained data to improve compression efficiency. The function `train_dict` can be used to generate a Zstandard dictionary.

compress(data, mode=ZstdCompressor.CONTINUE)

Compress *data* (a `bytes-like object`), returning a `bytes` object with compressed data if possible, or otherwise an empty `!bytes` object. Some of *data* may be buffered internally, for use in later calls to `!compress` and `~.flush`. The returned data should be concatenated with the output of any previous calls to `~.compress`.

The *mode* argument is a `ZstdCompressor` attribute, either `~.CONTINUE`, `~.FLUSH_BLOCK`, or `~.FLUSH_FRAME`.

When all data has been provided to the compressor, call the `~.flush` method to finish the compression process. If `~.compress` is called with *mode* set to `~.FLUSH_FRAME`, `~.flush` should not be called, as it would write out a new empty frame.

flush(mode=ZstdCompressor.FLUSH_FRAME)

Finish the compression process, returning a `bytes` object containing any data stored in the compressor's internal buffers.

The *mode* argument is a `ZstdCompressor` attribute, either `~.FLUSH_BLOCK`, or `~.FLUSH_FRAME`.

set_pledged_input_size(size)

Specify the amount of uncompressed data *size* that will be provided for the next frame. *size* will be written into the frame header of the next frame unless `CompressionParameter.content_size_flag` is `False` or `0`. A size of `0` means that the frame is empty. If *size* is `None`, the frame header will omit the frame size. Frames that include the uncompressed data size require less memory to decompress, especially at higher compression levels.

If `last_mode` is not `FLUSH_FRAME`, a `ValueError` is raised as the compressor is not at the start of a frame. If the pledged size does not match the actual size of data provided to `.compress`, future calls to `!compress` or `flush` may raise `ZstdError` and the last chunk of data may be lost.

After `flush` or `.compress` are called with mode `FLUSH_FRAME`, the next frame will not include the frame size into the header unless `!set_pledged_input_size` is called again.

CONTINUE

Collect more data for compression, which may or may not generate output immediately. This mode optimizes the compression ratio by maximizing the amount of data per block and frame.

FLUSH_BLOCK

Complete and write a block to the data stream. The data returned so far can be immediately decompressed. Past data can still be referenced in future blocks generated by calls to `~.compress`, improving compression.

FLUSH_FRAME

Complete and write out a frame. Future data provided to `~.compress` will be written into a new frame and *cannot* reference past data.

last_mode

The last mode passed to either `~.compress` or `~.flush`. The value can be one of `~.CONTINUE`, `~.FLUSH_BLOCK`, or `~.FLUSH_FRAME`. The initial value is `~.FLUSH_FRAME`, signifying that the compressor is at the start of a new frame.

Create a decompressor object, which can be used to decompress data incrementally.

For a more convenient way of decompressing an entire compressed stream at once, see the module-level function `decompress`.

The *options* argument is a Python dictionary containing advanced decompression parameters. The valid keys and values for compression parameters are documented as part of the `DecompressionParameter` documentation.

The *zstd_dict* argument is an instance of `ZstdDict` containing trained data used during compression. This must be the same Zstandard dictionary used during compression.

> This class does not transparently handle inputs containing multiple compressed frames, unlike the `decompress` function and `ZstdFile` class. To decompress a multi-frame input, you should use `decompress`, `ZstdFile` if working with a `file object`, or multiple `!ZstdDecompressor` instances.

decompress(data, max_length=-1)

Decompress *data* (a `bytes-like object`), returning uncompressed data as bytes. Some of *data* may be buffered internally, for use in later calls to `!decompress`. The returned data should be concatenated with the output of any previous calls to `!decompress`.

If *max_length* is non-negative, the method returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced (or EOF is reached), the `~.needs_input` attribute will be set to `False`. In this case, the next call to `~.decompress` may provide *data* as `b''` to obtain more of the output. The full content can thus be read like:

```
process_output(d.decompress(data, max_length))
while not d.eof and not d.needs_input:
    process_output(d.decompress(b"", max_length))
```

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

Attempting to decompress data after the end of a frame will raise a `EOFError`. Any data found after the end of the frame is ignored and saved in the `~.unused_data` attribute.

eof

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

unused_data

Data found after the end of the compressed stream.

Before the end of the stream is reached, this will be `b''`.

needs_input

`False` if the `.decompress` method can provide more decompressed data before requiring new compressed input.

Zstandard dictionaries

train_dict(samples, dict_size)

Train a Zstandard dictionary, returning a `ZstdDict` instance. Zstandard dictionaries enable more efficient compression of smaller sizes of data, which is traditionally difficult to compress due to less repetition. If you are compressing multiple similar groups of data (such as similar files), Zstandard dictionaries can improve compression ratios and speed significantly.

The *samples* argument (an iterable of `bytes` objects), is the population of samples used to train the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. The Zstandard documentation suggests an absolute maximum of no more than 100 KB, but the maximum can often be smaller depending on the data. Larger dictionaries generally slow down compression, but improve compression ratios. Smaller dictionaries lead to faster compression, but reduce the compression ratio.

finalize_dict(zstd_dict, /, samples, dict_size, level)

An advanced function for converting a "raw content" Zstandard dictionary into a regular Zstandard dictionary. "Raw content" dictionaries are a sequence of bytes that do not need to follow the structure of a normal Zstandard dictionary.

The *zstd_dict* argument is a `ZstdDict` instance with the `~ZstdDict.dict_content` containing the raw dictionary contents.

The *samples* argument (an iterable of `bytes` objects), contains sample data for generating the Zstandard dictionary.

The *dict_size* argument, an integer, is the maximum size (in bytes) the Zstandard dictionary should be. See `train_dict` for suggestions on the maximum dictionary size.

The *level* argument (an integer) is the compression level expected to be passed to the compressors using this dictionary. The dictionary information varies for each compression level, so tuning for the proper compression level can make compression more efficient.

A wrapper around Zstandard dictionaries. Dictionaries can be used to improve the compression of many small chunks of data. Use `train_dict` if you need to train a new dictionary from sample data.

The *dict_content* argument (a `bytes-like object`), is the already trained dictionary information.

The *is_raw* argument, a boolean, is an advanced parameter controlling the meaning of *dict_content* . `True` means *dict_content* is a "raw content" dictionary, without any format restrictions. `False` means *dict_content* is an ordinary Zstandard dictionary, created from Zstandard functions, for example, `train_dict` or the external `zstd` CLI.

When passing a `!ZstdDict` to a function, the `!as_digested_dict` and `!as_undigested_dict` attributes can control how the dictionary is loaded by passing them as the `zstd_dict` argument, for example, `compress(data, zstd_dict=zd.as_digested_dict)`. Digesting a dictionary is a costly operation that occurs when loading a Zstandard dictionary. When making multiple calls to compression or decompression, passing a digested dictionary will reduce the overhead of loading the dictionary.

> | | **Digested dictionary** | **Undigested dictionary** |
> |---|---|---|
> | Advanced parameters of the compressor which may be overridden by the dictionary's parameters | `window_log`, `hash_log`, `chain_log`, `search_log`, `min_match`, `target_length`, `strategy`, `enable_long_distance_matching`, `ldm_hash_log`, `ldm_min_match`, `ldm_bucket_size_log`, `ldm_hash_rate_log`, and some non-public parameters. | None |
> | `!ZstdDict` internally caches the dictionary | Yes. It's faster when loading a digested dictionary again with the same compression level. | No. If you wish to load an undigested dictionary multiple times, consider reusing a compressor object. |
>
> Difference for compression

If passing a `!ZstdDict` without any attribute, an undigested dictionary is passed by default when compressing and a digested dictionary is generated if necessary and passed by default when decompressing.

> dict_content
>
> The content of the Zstandard dictionary, a `bytes` object. It's the same as the *dict_content* argument in the `__init__` method. It can be used with other programs, such as the `zstd` CLI program.
>
> dict_id

> Identifier of the Zstandard dictionary, a non-negative int value.
>
> Non-zero means the dictionary is ordinary, created by Zstandard functions and following the Zstandard format.
>
> `0` means a "raw content" dictionary, free of any format restriction, used for advanced users.
>
> > [!NOTE] The meaning of `0` for `!ZstdDict.dict_id` is different from the `dictionary_id` attribute to the `get_frame_info` function.
>
> as_digested_dict
>
> Load as a digested dictionary.
>
> as_undigested_dict
>
> Load as an undigested dictionary.

Advanced parameter control

An `~enum.IntEnum` containing the advanced compression parameter keys that can be used when compressing data.

The `~.bounds` method can be used on any attribute to get the valid values for that parameter.

Parameters are optional; any omitted parameter will have its value selected automatically.

Example getting the lower and upper bound of `~.compression_level`:

```
lower, upper = CompressionParameter.compression_level.bounds()
```

Example setting the `~.window_log` to the maximum size:

```
_lower, upper = CompressionParameter.window_log.bounds()
options = {CompressionParameter.window_log: upper}
compress(b'venezuelan beaver cheese', options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a compression parameter. This method should be called on the attribute you wish to retrieve the bounds of. For example, to get the valid values for `~.compression_level`, one may check the result of `CompressionParameter.compression_level.bounds()`.

Both the lower and upper bounds are inclusive.

compression_level

A high-level means of setting other compression parameters that affect the speed and ratio of compressing data.

Regular compression levels are greater than `0`. Values greater than `20` are considered "ultra" compression and require more memory than other levels. Negative values can be used to trade off faster compression for worse compression ratios.

Setting the level to zero uses `COMPRESSION_LEVEL_DEFAULT`.

window_log

Maximum allowed back-reference distance the compressor can use when compressing data, expressed as power of two, `1 << window_log` bytes. This parameter greatly influences the memory usage of compression. Higher values require more memory but gain better compression values.

A value of zero causes the value to be selected automatically.

hash_log

Size of the initial probe table, as a power of two. The resulting memory usage is `1 << (hash_log+2)` bytes. Larger tables improve compression ratio of strategies <= `~Strategy.dfast`, and improve compression speed of strategies > `~Strategy.dfast`.

A value of zero causes the value to be selected automatically.

chain_log

Size of the multi-probe search table, as a power of two. The resulting memory usage is `1 << (chain_log+2)` bytes. Larger tables result in better and slower compression. This parameter has no effect for the `~Strategy.fast` strategy. It's still useful when using `~Strategy.dfast` strategy, in which case it defines a secondary probe table.

A value of zero causes the value to be selected automatically.

search_log

Number of search attempts, as a power of two. More attempts result in better and slower compression. This parameter is useless for `~Strategy.fast` and `~Strategy.dfast` strategies.

A value of zero causes the value to be selected automatically.

min_match

Minimum size of searched matches. Larger values increase compression and decompression speed, but decrease ratio. Note that Zstandard can still find matches of smaller size, it just tweaks its search algorithm to look for this size and larger. For all strategies < `~Strategy.btopt`, the effective minimum is `4`; for all strategies > `~Strategy.fast`, the effective maximum is `6`.

A value of zero causes the value to be selected automatically.

target_length

The impact of this field depends on the selected `Strategy`.

For strategies `~Strategy.btopt`, `~Strategy.btultra` and `~Strategy.btultra2`, the value is the length of a match considered "good enough" to stop searching. Larger values make compression ratios better, but compresses slower.

For strategy `~Strategy.fast`, it is the distance between match sampling. Larger values make compression faster, but with a worse compression ratio.

A value of zero causes the value to be selected automatically.

strategy

The higher the value of selected strategy, the more complex the compression technique used by zstd, resulting in higher compression ratios but slower compression.

`Strategy`

enable_long_distance_matching

Long distance matching can be used to improve compression for large inputs by finding large matches at greater distances. It increases memory usage and window size.

`True` or `1` enable long distance matching while `False` or `0` disable it.

Enabling this parameter increases default `~CompressionParameter.window_log` to 128 MiB except when expressly set to a different value. This setting is enabled by default if `!window_log` >= 128 MiB and the compression strategy >= `~Strategy.btopt` (compression level 16+).

ldm_hash_log

Size of the table for long distance matching, as a power of two. Larger values increase memory usage and compression ratio, but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_min_match

Minimum match size for long distance matcher. Larger or too small values can often decrease the compression ratio.

A value of zero causes the value to be selected automatically.

ldm_bucket_size_log

Log size of each bucket in the long distance matcher hash table for collision resolution. Larger values improve collision resolution but decrease compression speed.

A value of zero causes the value to be selected automatically.

ldm_hash_rate_log

Frequency of inserting/looking up entries into the long distance matcher hash table. Larger values improve compression speed. Deviating far from the default value will likely result in a compression ratio decrease.

A value of zero causes the value to be selected automatically.

content_size_flag

Write the size of the data to be compressed into the Zstandard frame header when known prior to compressing.

This flag only takes effect under the following scenarios:

- Calling `compress` for one-shot compression
- Providing all of the data to be compressed in the frame in a single `ZstdCompressor.compress` call, with the `ZstdCompressor.FLUSH_FRAME` mode.
- Calling `ZstdCompressor.set_pledged_input_size` with the exact amount of data that will be provided to the compressor prior to any calls to `ZstdCompressor.compress` for the current frame. `!ZstdCompressor.set_pledged_input_size` must be called for each new frame.

All other compression calls may not write the size information into the frame header.

`True` or `1` enable the content size flag while `False` or `0` disable it.

checksum_flag

A four-byte checksum using XXHash64 of the uncompressed content is written at the end of each frame. Zstandard's decompression code verifies the checksum. If there is a mismatch a `ZstdError` exception is raised.

`True` or `1` enable checksum generation while `False` or `0` disable it.

dict_id_flag

When compressing with a `ZstdDict`, the dictionary's ID is written into the frame header.

`True` or `1` enable storing the dictionary ID while `False` or `0` disable it.

nb_workers

Select how many threads will be spawned to compress in parallel. When `!nb_workers` > 0, enables multi-threaded compression, a value of `1` means "one-thread multi-threaded mode". More workers improve speed, but also increase memory usage and slightly reduce compression ratio.

A value of zero disables multi-threading.

job_size

Size of a compression job, in bytes. This value is enforced only when `~CompressionParameter.nb_workers` >= 1. Each compression job is completed in parallel, so this value can indirectly impact the number of active threads.

A value of zero causes the value to be selected automatically.

overlap_log

Sets how much data is reloaded from previous jobs (threads) for new jobs to be used by the look behind window during compression. This value is only used when `~CompressionParameter.nb_workers` >= 1. Acceptable values vary from 0 to 9.

> - 0 means dynamically set the overlap amount
> - 1 means no overlap
> - 9 means use a full window size from the previous job

Each increment halves/doubles the overlap size. "8" means an overlap of `window_size/2`, "7" means an overlap of `window_size/4`, etc.

An `~enum.IntEnum` containing the advanced decompression parameter keys that can be used when decompressing data. Parameters are optional; any omitted parameter will have its value selected automatically.

The `~.bounds` method can be used on any attribute to get the valid values for that parameter.

Example setting the `~.window_log_max` to the maximum size:

```
data = compress(b'Some very long buffer of bytes...')

_lower, upper = DecompressionParameter.window_log_max.bounds()

options = {DecompressionParameter.window_log_max: upper}
decompress(data, options=options)
```

bounds()

Return the tuple of int bounds, `(lower, upper)`, of a decompression parameter. This method should be called on the attribute you wish to retrieve the bounds of.

Both the lower and upper bounds are inclusive.

window_log_max

The base-two logarithm of the maximum size of the window used during decompression. This can be useful to limit the amount of memory used when decompressing data. A larger maximum window size leads to faster decompression.

A value of zero causes the value to be selected automatically.

An `~enum.IntEnum` containing strategies for compression. Higher-numbered strategies correspond to more complex and slower compression.

> The values of attributes of `!Strategy` are not necessarily stable across zstd versions. Only the ordering of the attributes may be relied upon. The attributes are listed below in order.

The following strategies are available:

fast

dfast

greedy

lazy

lazy2

btlazy2

btopt

btultra

btultra2

Miscellaneous

get_frame_info(frame_buffer)

Retrieve a `FrameInfo` object containing metadata about a Zstandard frame. Frames contain metadata related to the compressed data they hold.

Metadata related to a Zstandard frame.

decompressed_size

The size of the decompressed contents of the frame.

dictionary_id

An integer representing the Zstandard dictionary ID needed for decompressing the frame. `0` means the dictionary ID was not recorded in the frame header. This may mean that a Zstandard dictionary is not needed, or that the ID of a required dictionary was not recorded.

COMPRESSION_LEVEL_DEFAULT

The default compression level for Zstandard: `3`.

Information about the version of the zstd library in use is available through the following constants:

ZSTD_VERSION

The version string of the zstd library that was used for building the module. This may be different from the zstd library actually used at runtime, which is available as `zstd_version`.

3.16

zstd_version

The version string of the zstd library actually loaded by the interpreter.

ZSTD_VERSION_INFO

A named tuple containing the three components of the zstd library version that was used for building the module: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `zstd.ZSTD_VERSION_INFO[0]` is equivalent to `zstd.ZSTD_VERSION_INFO.major` and so on. This may be different from the zstd library actually used at runtime, which is available as `zstd_version_info`.

3.16

zstd_version_info

A named tuple containing the version of the zstd library actually loaded by the interpreter, with the same fields as `ZSTD_VERSION_INFO`.

3.16 It is now a named tuple.

Examples

Reading in a compressed file:

```
from compression import zstd

with zstd.open("file.zst") as f:
    file_content = f.read()
```

Creating a compressed file:

```
from compression import zstd

data = b"Insert Data Here"
with zstd.open("file.zst", "w") as f:
```

```
    f.write(data)
```

Compressing data in memory:

```
from compression import zstd

data_in = b"Insert Data Here"
data_out = zstd.compress(data_in)
```

Incremental compression:

```
from compression import zstd

comp = zstd.ZstdCompressor()
out1 = comp.compress(b"Some data\n")
out2 = comp.compress(b"Another piece of data\n")
out3 = comp.compress(b"Even more data\n")
out4 = comp.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

Writing compressed data to an already-open file:

```
from compression import zstd

with open("myfile", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with zstd.open(f, "w") as zstf:
        zstf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using compression parameters:

```
from compression import zstd

options = {
   zstd.CompressionParameter.checksum_flag: 1
}
with zstd.open("file.zst", "w", options=options) as f:
    f.write(b"Mind if I squeeze in?")
```

□□□ #2
□ □□□ □□□ 2026-10-31 17:05:57 UTC
□ □□□ □□□ 2026-10-31 17:06:01 UTC