12.1. `zlib` — Compression compatible with gzip

For applications that require data compression, the functions in this module allow compression and decompression, using the zlib library. The zlib library has its own home page at http://www.zlib.net. There are known incompatibilities between the Python module and versions of the zlib library earlier than 1.1.3; 1.1.3 has a security vulnerability, so we recommend using 1.1.4 or later.

zlib's functions have many options and often need to be used in a particular order. This documentation doesn't attempt to cover all of the permutations; consult the zlib manual at http://www.zlib.net/manual.html for authoritative information.

For reading and writing `.gz` files see the `gzip` module.

The available exception and functions in this module are:

exception `zlib.`**`error`**

Exception raised on compression and decompression errors.

`zlib.`**`adler32`**(*data*[, *value*])

Computes an Adler-32 checksum of *data*. (An Adler-32 checksum is almost as reliable as a CRC32 but can be computed much more quickly.) If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

This function always returns an integer object.

Note: To generate the same numeric value across all Python versions and platforms use adler32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

Changed in version 2.6: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value is signed on some platforms and unsigned on others.

Changed in version 3.0: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

`zlib.`**`compress`**(*string*[, *level*])

Compresses the data in *string*, returning a string contained compressed data. *level* is an integer from `0` to `9` controlling the level of compression; `1` is fastest and produces the least compression, `9` is slowest and produces the most. `0` is no compression.

The default value is `6`. Raises the `error` exception if any error occurs.

`zlib.`**`compressobj`**([*level*[, *method*[, *wbits*[, *memlevel*[, *strategy*]]]]])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once. *level* is an integer from `0` to `9` or `-1`, controlling the level of compression; `1` is fastest and produces the least compression, `9` is slowest and produces the most. `0` is no compression. The default value is `-1` (Z_DEFAULT_COMPRESSION). Z_DEFAULT_COMPRESSION represents a default compromise between speed and compression (currently equivalent to level 6).

method is the compression algorithm. Currently, the only supported value is `DEFLATED`.

wbits is the base two logarithm of the size of the window buffer. This should be an integer from `8` to `15`. Higher values give better compression, but use more memory. The default is 15.

memlevel controls the amount of memory used for internal compression state. Valid values range from `1` to `9`. Higher values using more memory, but are faster and produce smaller output. The default is 8.

strategy is used to tune the compression algorithm. Possible values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, and `Z_HUFFMAN_ONLY`. The default is `Z_DEFAULT_STRATEGY`.

`zlib.`**`crc32`**(*data*[, *value*])

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. If *value* is present, it is used as the starting value of the checksum; otherwise, a fixed default value is used. This allows computing a running checksum over the concatenation of several inputs. The algorithm is not cryptographically strong, and should not be used for authentication or digital signatures. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm.

This function always returns an integer object.

Note: To generate the same numeric value across all Python versions and platforms use crc32(data) & 0xffffffff. If you are only using the checksum in packed binary format this is not necessary as the return value is the correct 32bit binary representation regardless of sign.

Changed in version 2.6: The return value is in the range [-2**31, 2**31-1] regardless of platform. In older versions the value would be signed on some platforms and unsigned on others.

Changed in version 3.0: The return value is unsigned and in the range [0, 2**32-1] regardless of platform.

`zlib.`**`decompress`**(*string*[, *wbits*[, *bufsize*]])

Decompresses the data in *string*, returning a string containing the uncompressed data. The *wbits* parameter controls the size of the window buffer, and is discussed further below. If *bufsize* is given, it is used as the initial size of the output buffer. Raises the `error` exception if any error occurs.

The absolute value of *wbits* is the base two logarithm of the size of the history buffer (the “window size”) used when compressing data. Its absolute value should be between 8 and 15 for the most recent versions of the zlib library, larger values resulting in better compression at the expense of greater memory usage. When decompressing a stream, *wbits* must not be smaller than the size originally used to compress the stream; using a too-small value will result in an exception. The default value is therefore the highest value, 15. When *wbits* is negative, the standard **gzip** header is suppressed.

bufsize is the initial size of the buffer used to hold decompressed data. If more space is required, the buffer size will be increased as needed, so you don’t have to get this value exactly right; tuning it will only save a few calls to `malloc()`. The default size is 16384.

`zlib.`**`decompressobj`**([*wbits*])

Returns a decompression object, to be used for decompressing data streams that won’t fit into memory at once. The *wbits* parameter controls the size of the window buffer.

Compression objects support the following methods:

`Compress.`**`compress`**(*string*)

Compress *string*, returning a string containing compressed data for at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the `compress()` method. Some input may be kept in internal buffers for later processing.

`Compress.`**`flush`**([*mode*])

All pending input is processed, and a string containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, or `Z_FINISH`, defaulting to `Z_FINISH`. `Z_SYNC_FLUSH` and `Z_FULL_FLUSH` allow compressing further strings of data, while `Z_FINISH` finishes the compressed stream and prevents compressing any more data. After calling `flush()` with *mode* set to `Z_FINISH`, the `compress()` method cannot be called again; the only realistic action is to delete the object.

`Compress.`**`copy`**()

Returns a copy of the compression object. This can be used to efficiently compress a set of data that share a common initial prefix.

New in version 2.5.

Decompression objects support the following methods, and two attributes:

`Decompress.`**`unused_data`**

A string which contains any bytes past the end of the compressed data. That is, this remains `""` until the last byte that contains compression data is available. If the whole string turned out to contain compressed data, this is `""`, the empty string.

The only way to determine where a string of compressed data ends is by actually decompressing it. This means that when compressed data is contained part of a larger file, you can only find the end of it by reading data and feeding it followed by some non-empty string into a decompression object's `decompress()` method until the `unused_data` attribute is no longer the empty string.

`Decompress.`**`unconsumed_tail`**

A string that contains any data that was not consumed by the last `decompress()` call because it exceeded the limit for the uncompressed data buffer. This data has not yet been seen by the zlib machinery, so you must feed it (possibly with further data concatenated to it) back to a subsequent `decompress()` method call in order to get correct output.

`Decompress.`**`decompress`**(*string*[, *max_length*])

Decompress *string*, returning a string containing the uncompressed data corresponding to at least part of the data in *string*. This data should be concatenated to the output produced by any preceding calls to the `decompress()` method. Some of the input data may be preserved in internal buffers for later processing.

If the optional parameter *max_length* is non-zero then the return value will be no longer than *max_length*. This may mean that not all of the compressed input can be processed; and unconsumed data will be stored in the attribute `unconsumed_tail`. This string must be passed to a subsequent call to `decompress()` if decompression is to continue. If *max_length* is not supplied then the whole input is decompressed, and `unconsumed_tail` is an empty string.

`Decompress.`**`flush`**([*length*])

All pending input is processed, and a string containing the remaining uncompressed output is returned. After calling `flush()`, the `decompress()` method cannot be called again; the only realistic action is to delete the object.

The optional parameter *length* sets the initial size of the output buffer.

`Decompress.`**`copy`**()

Returns a copy of the decompression object. This can be used to save the state of the decompressor midway through the data stream in order to speed up random seeks into the stream at a future point.

New in version 2.5.

See also:

Module `gzip`

Reading and writing **gzip**-format files.

http://www.zlib.net
: The zlib library home page.

http://www.zlib.net/manual.html
: The zlib manual explains the semantics and usage of the library’s many functions.