CHAPTER

THIRTEEN

DATA COMPRESSION AND ARCHIVING

The modules described in this chapter support data compression with the zlib, gzip, bzip2 and lzma algorithms, and the creation of ZIP- and tar-format archives. See also *Archiving operations* provided by the *shutil* module.

13.1 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.) The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 1 is used. Passing in *value* 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.

Changed in version 3.0: Always returns an unsigned value. To generate the same numeric value across all Python versions and platforms, use `adler32(data) & 0xffffffff`.

zlib.compress(*data*, *level=-1*)
Compresses the bytes in *data*, returning a bytes object containing compressed data. *level* is an integer from `0` to `9` or `-1` controlling the level of compression; `1` (Z_BEST_SPEED) is fastest and produces the least compression, `9` (Z_BEST_COMPRESSION) is slowest and produces the most. `0` (Z_NO_COMPRESSION) 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). Raises the *error* exception if any error occurs.

Changed in version 3.6: *level* can now be used as a keyword parameter.

zlib.compressobj(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

Returns a compression object, to be used for compressing data streams that won't fit into memory at once.

level is the compression level – an integer from `0` to `9` or `-1`. A value of `1` (Z_BEST_SPEED) is fastest and produces the least compression, while a value of `9` (Z_BEST_COMPRESSION) is slowest and produces the most. `0` (Z_NO_COMPRESSION) 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`.

The *wbits* argument controls the size of the history buffer (or the "window size") used when compressing data, and whether a header and trailer is included in the output. It can take several ranges of values, defaulting to `15` (MAX_WBITS):

- +9 to +15: The base-two logarithm of the window size, which therefore ranges between 512 and 32768. Larger values produce better compression at the expense of greater memory usage. The resulting output will include a zlib-specific header and trailer.
- −9 to −15: Uses the absolute value of *wbits* as the window size logarithm, while producing a raw output stream with no header or trailing checksum.
- +25 to +31 = 16 + (9 to 15): Uses the low 4 bits of the value as the window size logarithm, while including a basic **gzip** header and trailing checksum in the output.

The *memLevel* argument controls the amount of memory used for the internal compression state. Valid values range from `1` to `9`. Higher values use more memory, but are faster and produce smaller output.

strategy is used to tune the compression algorithm. Possible values are `Z_DEFAULT_STRATEGY`, `Z_FILTERED`, `Z_HUFFMAN_ONLY`, `Z_RLE` (zlib 1.2.0.1) and `Z_FIXED` (zlib 1.2.2.2).

zdict is a predefined compression dictionary. This is a sequence of bytes (such as a *bytes* object) containing subsequences that are expected to occur frequently in the data that is to be compressed. Those subsequences that are expected to be most common should come at the end of the dictionary.

Changed in version 3.3: Added the *zdict* parameter and keyword argument support.

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

Computes a CRC (Cyclic Redundancy Check) checksum of *data*. The result is an unsigned 32-bit integer. If *value* is present, it is used as the starting value of the checksum; otherwise, a default value of 0 is used. Passing in *value* 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.

Changed in version 3.0: Always returns an unsigned value. To generate the same numeric value across all Python versions and platforms, use `crc32(data) & 0xffffffff`.

zlib.decompress(*data*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

Decompresses the bytes in *data*, returning a bytes object containing the uncompressed data. The *wbits* parameter depends on the format of *data*, 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 *wbits* parameter controls the size of the history buffer (or "window size"), and what header and trailer format is expected. It is similar to the parameter for *compressobj()*, but accepts more ranges of values:

- +8 to +15: The base-two logarithm of the window size. The input must include a zlib header and trailer.
- 0: Automatically determine the window size from the zlib header. Only supported since zlib 1.2.3.5.
- −8 to −15: Uses the absolute value of *wbits* as the window size logarithm. The input must be a raw stream with no header or trailer.
- +24 to +31 = 16 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm. The input must include a gzip header and trailer.
- +40 to +47 = 32 + (8 to 15): Uses the low 4 bits of the value as the window size logarithm, and automatically accepts either the zlib or gzip format.

When decompressing a stream, the window size must not be smaller than the size originally used to compress the stream; using a too-small value may result in an *error* exception. The default *wbits* value corresponds to the largest window size and requires a zlib header and trailer to be included.

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()`.

Changed in version 3.6: *wbits* and *bufsize* can be used as keyword arguments.

`zlib.decompressobj`(*wbits=MAX_WBITS*[, *zdict*])
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 history buffer (or the "window size"), and what header and trailer format is expected. It has the same meaning as *described for decompress()*.

The *zdict* parameter specifies a predefined compression dictionary. If provided, this must be the same dictionary as was used by the compressor that produced the data that is to be decompressed.

Note: If *zdict* is a mutable object (such as a *bytearray*), you must not modify its contents between the call to *decompressobj()* and the first call to the decompressor's `decompress()` method.

Changed in version 3.3: Added the *zdict* parameter.

Compression objects support the following methods:

`Compress.compress`(*data*)
Compress *data*, returning a bytes object containing compressed data for at least part of the data in *data*. 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 bytes object containing the remaining compressed output is returned. *mode* can be selected from the constants `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH`, `Z_BLOCK` (zlib 1.2.3.4), or `Z_FINISH`, defaulting to `Z_FINISH`. Except `Z_FINISH`, all constants allow compressing further bytestrings 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.

Decompression objects support the following methods and attributes:

Decompress.**unused_data**
: A bytes object which contains any bytes past the end of the compressed data. That is, this remains `b""` until the last byte that contains compression data is available. If the whole bytestring turned out to contain compressed data, this is `b""`, an empty bytes object.

Decompress.**unconsumed_tail**
: A bytes object 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.**eof**
: A boolean indicating whether the end of the compressed data stream has been reached.

  This makes it possible to distinguish between a properly-formed compressed stream, and an incomplete or truncated one.

  New in version 3.3.

Decompress.**decompress**(*data*, *max_length=0*)
: Decompress *data*, returning a bytes object 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 bytestring must be passed to a subsequent call to *decompress()* if decompression is to continue. If *max_length* is zero then the whole input is decompressed, and *unconsumed_tail* is empty.

  Changed in version 3.6: *max_length* can be used as a keyword argument.

Decompress.**flush**([*length*])
: All pending input is processed, and a bytes object 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.

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

zlib.**ZLIB_VERSION**
: The version string of the zlib library that was used for building the module. This may be different from the zlib library actually used at runtime, which is available as *ZLIB_RUNTIME_VERSION*.

zlib.**ZLIB_RUNTIME_VERSION**
: The version string of the zlib library actually loaded by the interpreter.

  New in version 3.3.

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.

13.2 gzip — Support for gzip files

Source code: Lib/gzip.py

This module provides a simple interface to compress and decompress files just like the GNU programs **gzip** and **gunzip** would.

The data compression is provided by the *zlib* module.

The *gzip* module provides the *GzipFile* class, as well as the *open()*, *compress()* and *decompress()* convenience functions. The *GzipFile* class reads and writes **gzip**-format files, automatically compressing or decompressing the data so that it looks like an ordinary *file object*.

Note that additional file formats which can be decompressed by the **gzip** and **gunzip** programs, such as those produced by **compress** and **pack**, are not supported by this module.

The module defines the following items:

gzip.open(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
Open a gzip-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

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

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

For binary mode, this function is equivalent to the *GzipFile* constructor: `GzipFile(filename, mode, compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

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

Changed in version 3.3: Added support for *filename* being a file object, support for text mode, and the *encoding*, *errors* and *newline* arguments.

Changed in version 3.4: Added support for the `'x'`, `'xb'` and `'xt'` modes.

Changed in version 3.6: Accepts a *path-like object*.

class gzip.GzipFile(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)
Constructor for the *GzipFile* class, which simulates most of the methods of a *file object*, with the exception of the `truncate()` method. At least one of *fileobj* and *filename* must be given a non-trivial value.

The new class instance is based on *fileobj*, which can be a regular file, an *io.BytesIO* object, or any other object which simulates a file. It defaults to `None`, in which case *filename* is opened to provide a file object.

When *fileobj* is not `None`, the *filename* argument is only used to be included in the **gzip** file header, which may include the original filename of the uncompressed file. It defaults to the filename of *fileobj*, if discernible; otherwise, it defaults to the empty string, and in this case the original filename is not included in the header.

The *mode* argument can be any of `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'`, or `'xb'`, depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is `'rb'`.

Note that the file is always opened in binary mode. To open a compressed file in text mode, use *open()* (or wrap your *GzipFile* with an *io.TextIOWrapper*).

The *compresslevel* argument is an integer from `0` to `9` controlling the level of compression; `1` is fastest and produces the least compression, and `9` is slowest and produces the most compression. `0` is no compression. The default is `9`.

The *mtime* argument is an optional numeric timestamp to be written to the last modification time field in the stream when compressing. It should only be provided in compression mode. If omitted or `None`, the current time is used. See the *mtime* attribute for more details.

Calling a *GzipFile* object's `close()` method does not close *fileobj*, since you might wish to append more material after the compressed data. This also allows you to pass an *io.BytesIO* object opened for writing as *fileobj*, and retrieve the resulting memory buffer using the *io.BytesIO* object's *getvalue()* method.

GzipFile supports the *io.BufferedIOBase* interface, including iteration and the `with` statement. Only the `truncate()` method isn't implemented.

GzipFile also provides the following method and attribute:

peek(*n*)
Read *n* uncompressed bytes without advancing the file position. At most one single read on the compressed stream is done to satisfy the call. The number of bytes returned may be more or less than requested.

Note: While calling *peek()* does not change the file position of the *GzipFile*, it may change the position of the underlying file object (e.g. if the *GzipFile* was constructed with the *fileobj* parameter).

New in version 3.2.

mtime
When decompressing, the value of the last modification time field in the most recently read header may be read from this attribute, as an integer. The initial value before reading any headers is `None`.

All **gzip** compressed streams are required to contain this timestamp field. Some programs, such as **gunzip**, make use of the timestamp. The format is the same as the return value of *time.time()* and the *st_mtime* attribute of the object returned by *os.stat()*.

Changed in version 3.1: Support for the `with` statement was added, along with the *mtime* constructor argument and *mtime* attribute.

Changed in version 3.2: Support for zero-padded and unseekable files was added.

Changed in version 3.3: The *io.BufferedIOBase.read1()* method is now implemented.

Changed in version 3.4: Added support for the `'x'` and `'xb'` modes.

Changed in version 3.5: Added support for writing arbitrary *bytes-like objects*. The *read()* method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

gzip.compress(*data*, *compresslevel=9*)
Compress the *data*, returning a *bytes* object containing the compressed data. *compresslevel* has the same meaning as in the *GzipFile* constructor above.

New in version 3.2.

gzip.decompress(*data*)
Decompress the *data*, returning a *bytes* object containing the uncompressed data.

New in version 3.2.

13.2.1 Examples of usage

Example of how to read a compressed file:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

Example of how to create a compressed GZIP file:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

Example of how to GZIP compress an existing file:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

Example of how to GZIP compress a binary string:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

See also:

Module *zlib* The basic data compression module needed to support the **gzip** file format.

13.3 bz2 — Support for bzip2 compression

Source code: Lib/bz2.py

This module provides a comprehensive interface for compressing and decompressing data using the bzip2 compression algorithm.

The *bz2* module contains:

- The *open()* function and *BZ2File* class for reading and writing compressed files.
- The *BZ2Compressor* and *BZ2Decompressor* classes for incremental (de)compression.
- The *compress()* and *decompress()* functions for one-shot (de)compression.

All of the classes in this module may safely be accessed from multiple threads.

13.3.1 (De)compression of files

bz2.open(*filename*, *mode='r'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
Open a bzip2-compressed file in binary or text mode, returning a *file object*.

As with the constructor for *BZ2File*, the *filename* argument can be an actual filename (a *str* or *bytes* object), or an existing file object to read from or write to.

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

The *compresslevel* argument is an integer from 1 to 9, as for the *BZ2File* constructor.

For binary mode, this function is equivalent to the *BZ2File* constructor: `BZ2File(filename, mode, compresslevel=compresslevel)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

For text mode, a *BZ2File* object is created, and wrapped in an *io.TextIOWrapper* instance with the specified encoding, error handling behavior, and line ending(s).

New in version 3.3.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: Accepts a *path-like object*.

`class bz2.BZ2File(`*filename*, *mode='r'*, *buffering=None*, *compresslevel=9*`)`

Open a bzip2-compressed file in binary mode.

If *filename* is a *str* or *bytes* object, open the named file directly. Otherwise, *filename* should be a *file object*, which will be used to read or write the compressed data.

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

If *filename* 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'`.

The *buffering* argument is ignored. Its use is deprecated.

If *mode* is `'w'` or `'a'`, *compresslevel* can be a number between `1` and `9` specifying the level of compression: `1` produces the least compression, and `9` (default) produces the most compression.

If *mode* is `'r'`, the input file may be the concatenation of multiple compressed streams.

BZ2File provides all of the members specified by the *io.BufferedIOBase*, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

BZ2File also provides the following method:

`peek(`[*n*]`)`

Return buffered data without advancing the file position. At least one byte of data will be returned (unless at EOF). The exact number of bytes returned is unspecified.

Note: While calling *peek()* does not change the file position of the *BZ2File*, it may change the position of the underlying file object (e.g. if the *BZ2File* was constructed by passing a file object for *filename*).

New in version 3.3.

Changed in version 3.1: Support for the `with` statement was added.

Changed in version 3.3: The `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` and `readinto()` methods were added.

Changed in version 3.3: Support was added for *filename* being a *file object* instead of an actual filename.

Changed in version 3.3: The `'a'` (append) mode was added, along with support for reading multi-stream files.

Changed in version 3.4: The `'x'` (exclusive creation) mode was added.

Changed in version 3.5: The *read()* method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

13.3.2 Incremental (de)compression

class bz2.BZ2Compressor(*compresslevel=9*)

Create a new compressor object. This object may be used to compress data incrementally. For one-shot compression, use the *compress()* function instead.

compresslevel, if given, must be a number between `1` and `9`. The default is `9`.

compress(*data*)

Provide data to the compressor object. Returns a chunk of compressed data if possible, or an empty byte string otherwise.

When you have finished providing data to the compressor, call the *flush()* method to finish the compression process.

flush()

Finish the compression process. Returns the compressed data left in internal buffers.

The compressor object may not be used after this method has been called.

class bz2.BZ2Decompressor

Create a new decompressor object. This object may be used to decompress data incrementally. For one-shot compression, use the *decompress()* function instead.

Note: This class does not transparently handle inputs containing multiple compressed streams, unlike *decompress()* and *BZ2File*. If you need to decompress a multi-stream input with *BZ2Decompressor*, you must use a new decompressor for each stream.

---

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 nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide *data* as `b''` to obtain more of the output.

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 stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

Changed in version 3.5: Added the *max_length* parameter.

eof

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

New in version 3.3.

unused_data

Data found after the end of the compressed stream.

If this attribute is accessed before the end of the stream has been reached, its value will be `b''`.

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

New in version 3.5.

13.3.3 One-shot (de)compression

bz2.compress(*data*, *compresslevel=9*)
Compress *data*.

compresslevel, if given, must be a number between `1` and `9`. The default is `9`.

For incremental compression, use a *BZ2Compressor* instead.

bz2.decompress(*data*)
Decompress *data*.

If *data* is the concatenation of multiple compressed streams, decompress all of the streams.

For incremental decompression, use a *BZ2Decompressor* instead.

Changed in version 3.3: Support for multi-stream inputs was added.

13.4 `lzma` — Compression using the LZMA algorithm

New in version 3.3.

Source code: Lib/lzma.py

This module provides classes and convenience functions for compressing and decompressing data using the LZMA compression algorithm. Also included is a file interface supporting the `.xz` and legacy `.lzma` file formats used by the **xz** utility, as well as raw compressed streams.

The interface provided by this module is very similar to that of the *bz2* module. However, note that *LZMAFile* is *not* thread-safe, unlike *bz2.BZ2File*, so if you need to use a single *LZMAFile* instance from multiple threads, it is necessary to protect it with a lock.

exception lzma.LZMAError
This exception is raised when an error occurs during compression or decompression, or while initializing the compressor/decompressor state.

13.4.1 Reading and writing compressed files

lzma.open(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)
Open an LZMA-compressed file in binary or text mode, returning a *file object*.

The *filename* argument can be either an actual file name (given as a *str*, *bytes* or *path-like* 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 any of `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` or `"ab"` for binary mode, or `"rt"`, `"wt"`, `"xt"`, or `"at"` for text mode. The default is `"rb"`.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for *LZMADecompressor*. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for *LZMACompressor*.

For binary mode, this function is equivalent to the *LZMAFile* constructor: `LZMAFile(filename, mode, ...)`. In this case, the *encoding*, *errors* and *newline* arguments must not be provided.

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

Changed in version 3.4: Added support for the `"x"`, `"xb"` and `"xt"` modes.

Changed in version 3.6: Accepts a *path-like object*.

class lzma.LZMAFile(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

Open an LZMA-compressed file in binary mode.

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

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

If *filename* 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 opening a file for reading, the input file may be the concatenation of multiple separate compressed streams. These are transparently decoded as a single logical stream.

When opening a file for reading, the *format* and *filters* arguments have the same meanings as for *LZMADecompressor*. In this case, the *check* and *preset* arguments should not be used.

When opening a file for writing, the *format*, *check*, *preset* and *filters* arguments have the same meanings as for *LZMACompressor*.

LZMAFile supports all the members specified by *io.BufferedIOBase*, except for `detach()` and `truncate()`. Iteration and the `with` statement are supported.

The following method is 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).

Note: While calling *peek()* does not change the file position of the *LZMAFile*, it may change the position of the underlying file object (e.g. if the *LZMAFile* was constructed by passing a file object for *filename*).

Changed in version 3.4: Added support for the `"x"` and `"xb"` modes.

Changed in version 3.5: The *read()* method now accepts an argument of `None`.

Changed in version 3.6: Accepts a *path-like object*.

13.4.2 Compressing and decompressing data in memory

class lzma.LZMACompressor(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

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 *compress()*.

The *format* argument specifies what container format should be used. Possible values are:

- **FORMAT_XZ: The .xz container format.** This is the default format.
- **FORMAT_ALONE: The legacy .lzma container format.** This format is more limited than .xz – it does not support integrity checks or multiple filters.
- **FORMAT_RAW: A raw data stream, not using any container format.** This format specifier does not support integrity checks, and requires that you always specify a custom filter chain (for both compression and decompression). Additionally, data compressed in this manner cannot be decompressed using FORMAT_AUTO (see *LZMADecompressor*).

The *check* argument specifies the type of integrity check to include in the compressed data. This check is used when decompressing, to ensure that the data has not been corrupted. Possible values are:

- CHECK_NONE: No integrity check. This is the default (and the only acceptable value) for FORMAT_ALONE and FORMAT_RAW.
- CHECK_CRC32: 32-bit Cyclic Redundancy Check.
- CHECK_CRC64: 64-bit Cyclic Redundancy Check. This is the default for FORMAT_XZ.
- CHECK_SHA256: 256-bit Secure Hash Algorithm.

If the specified check is not supported, an *LZMAError* is raised.

The compression settings can be specified either as a preset compression level (with the *preset* argument), or in detail as a custom filter chain (with the *filters* argument).

The *preset* argument (if provided) should be an integer between 0 and 9 (inclusive), optionally OR-ed with the constant PRESET_EXTREME. If neither *preset* nor *filters* are given, the default behavior is to use PRESET_DEFAULT (preset level 6). Higher presets produce smaller output, but make the compression process slower.

Note: In addition to being more CPU-intensive, compression with higher presets also requires much more memory (and produces output that needs more memory to decompress). With preset 9 for example, the overhead for an *LZMACompressor* object can be as high as 800 MiB. For this reason, it is generally best to stick with the default preset.

The *filters* argument (if provided) should be a filter chain specifier. See *Specifying custom filter chains* for details.

compress(*data*)
Compress *data* (a *bytes* object), returning a *bytes* object containing compressed data for at least part of the input. 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()*.

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

The compressor cannot be used after this method has been called.

class lzma.LZMADecompressor(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
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 *decompress()*.

The *format* argument specifies the container format that should be used. The default is FORMAT_AUTO, which can decompress both .xz and .lzma files. Other possible values are FORMAT_XZ, FORMAT_ALONE, and FORMAT_RAW.

The *memlimit* argument specifies a limit (in bytes) on the amount of memory that the decompressor can use. When this argument is used, decompression will fail with an *LZMAError* if it is not possible to decompress the input within the given memory limit.

The *filters* argument specifies the filter chain that was used to create the stream being decompressed. This argument is required if *format* is `FORMAT_RAW`, but should not be used for other formats. See *Specifying custom filter chains* for more information about filter chains.

Note: This class does not transparently handle inputs containing multiple compressed streams, unlike *decompress()* and *LZMAFile*. To decompress a multi-stream input with *LZMADecompressor*, you must create a new decompressor for each stream.

---

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 nonnegative, returns at most *max_length* bytes of decompressed data. If this limit is reached and further output can be produced, the *needs_input* attribute will be set to `False`. In this case, the next call to *decompress()* may provide *data* as `b''` to obtain more of the output.

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 stream is reached raises an *EOFError*. Any data found after the end of the stream is ignored and saved in the *unused_data* attribute.

Changed in version 3.5: Added the *max_length* parameter.

check

The ID of the integrity check used by the input stream. This may be `CHECK_UNKNOWN` until enough of the input has been decoded to determine what integrity check it uses.

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 uncompressed input.

New in version 3.5.

lzma.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

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

See *LZMACompressor* above for a description of the *format*, *check*, *preset* and *filters* arguments.

lzma.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)

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

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

See *LZMADecompressor* above for a description of the *format*, *memlimit* and *filters* arguments.

13.4.3 Miscellaneous

`lzma.is_check_supported(`*check*`)`
Returns true if the given integrity check is supported on this system.

`CHECK_NONE` and `CHECK_CRC32` are always supported. `CHECK_CRC64` and `CHECK_SHA256` may be unavailable if you are using a version of **liblzma** that was compiled with a limited feature set.

13.4.4 Specifying custom filter chains

A filter chain specifier is a sequence of dictionaries, where each dictionary contains the ID and options for a single filter. Each dictionary must contain the key `"id"`, and may contain additional keys to specify filter-dependent options. Valid filter IDs are as follows:

- **Compression filters:**
  - `FILTER_LZMA1` (for use with `FORMAT_ALONE`)
  - `FILTER_LZMA2` (for use with `FORMAT_XZ` and `FORMAT_RAW`)
- **Delta filter:**
  - `FILTER_DELTA`
- **Branch-Call-Jump (BCJ) filters:**
  - `FILTER_X86`
  - `FILTER_IA64`
  - `FILTER_ARM`
  - `FILTER_ARMTHUMB`
  - `FILTER_POWERPC`
  - `FILTER_SPARC`

A filter chain can consist of up to 4 filters, and cannot be empty. The last filter in the chain must be a compression filter, and any other filters must be delta or BCJ filters.

Compression filters support the following options (specified as additional entries in the dictionary representing the filter):

- `preset`: A compression preset to use as a source of default values for options that are not specified explicitly.
- `dict_size`: Dictionary size in bytes. This should be between 4 KiB and 1.5 GiB (inclusive).
- `lc`: Number of literal context bits.
- `lp`: Number of literal position bits. The sum `lc + lp` must be at most 4.
- `pb`: Number of position bits; must be at most 4.
- `mode`: `MODE_FAST` or `MODE_NORMAL`.
- `nice_len`: What should be considered a "nice length" for a match. This should be 273 or less.
- `mf`: What match finder to use – `MF_HC3`, `MF_HC4`, `MF_BT2`, `MF_BT3`, or `MF_BT4`.
- `depth`: Maximum search depth used by match finder. 0 (default) means to select automatically based on other filter options.

The delta filter stores the differences between bytes, producing more repetitive input for the compressor in certain circumstances. It supports one option, `dist`. This indicates the distance between bytes to be subtracted. The default is 1, i.e. take the differences between adjacent bytes.

The BCJ filters are intended to be applied to machine code. They convert relative branches, calls and jumps in the code to use absolute addressing, with the aim of increasing the redundancy that can be exploited by the compressor. These filters support one option, `start_offset`. This specifies the address that should be mapped to the beginning of the input data. The default is 0.

13.4.5 Examples

Reading in a compressed file:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

Creating a compressed file:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

Compressing data in memory:

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

Incremental compression:

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

Writing compressed data to an already-open file:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

Creating a compressed file using a custom filter chain:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile — Work with ZIP archives

Source code: Lib/zipfile.py

The ZIP file format is a common archive and compression standard. This module provides tools to create, read, write, append, and list a ZIP file. Any advanced use of this module will require an understanding of the format, as defined in PKZIP Application Note.

This module does not currently handle multi-disk ZIP files. It can handle ZIP files that use the ZIP64 extensions (that is ZIP files that are more than 4 GiB in size). It supports decryption of encrypted files in ZIP archives, but it currently cannot create an encrypted file. Decryption is extremely slow as it is implemented in native Python rather than C.

The module defines the following items:

exception zipfile.BadZipFile
The error raised for bad ZIP files.

New in version 3.2.

exception zipfile.BadZipfile
Alias of *BadZipFile*, for compatibility with older Python versions.

Deprecated since version 3.2.

exception zipfile.LargeZipFile
The error raised when a ZIP file would require ZIP64 functionality but that has not been enabled.

class zipfile.ZipFile
The class for reading and writing ZIP files. See section *ZipFile Objects* for constructor details.

class zipfile.PyZipFile
Class for creating ZIP archives containing Python libraries.

class zipfile.ZipInfo(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)
Class used to represent information about a member of an archive. Instances of this class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Most users of the *zipfile* module will not need to create these, but only use those created by this module. *filename* should be the full name of the archive member, and *date_time* should be a tuple containing six fields which describe the time of the last modification to the file; the fields are described in section *ZipInfo Objects*.

zipfile.is_zipfile(*filename*)
Returns `True` if *filename* is a valid ZIP file based on its magic number, otherwise returns `False`. *filename* may be a file or file-like object too.

Changed in version 3.1: Support for file and file-like objects.

zipfile.ZIP_STORED
The numeric constant for an uncompressed archive member.

zipfile.ZIP_DEFLATED
The numeric constant for the usual ZIP compression method. This requires the *zlib* module.

zipfile.ZIP_BZIP2
The numeric constant for the BZIP2 compression method. This requires the *bz2* module.

New in version 3.3.

zipfile.ZIP_LZMA
The numeric constant for the LZMA compression method. This requires the *lzma* module.

New in version 3.3.

Note: The ZIP file format specification has included support for bzip2 compression since 2001, and for LZMA compression since 2006. However, some tools (including older Python releases) do not support these compression methods, and may either refuse to process the ZIP file altogether, or fail to extract individual files.

See also:

PKZIP Application Note Documentation on the ZIP file format by Phil Katz, the creator of the format and algorithms used.

Info-ZIP Home Page Information about the Info-ZIP project's ZIP archive programs and development libraries.

13.5.1 ZipFile Objects

class zipfile.ZipFile(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*)

Open a ZIP file, where *file* can be a path to a file (a string), a file-like object or a *path-like object*.

The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If *mode* is `'x'` and *file* refers to an existing file, a *FileExistsError* will be raised. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If *mode* is `'a'` and the file does not exist at all, it is created. If *mode* is `'r'` or `'a'`, the file should be seekable.

compression is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA*; unrecognized values will cause *NotImplementedError* to be raised. If *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* is specified but the corresponding module (*zlib*, *bz2* or *lzma*) is not available, *RuntimeError* is raised. The default is *ZIP_STORED*.

If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is `false` *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

The *compresslevel* parameter controls the compression level to use when writing files to the archive. When using *ZIP_STORED* or *ZIP_LZMA* it has no effect. When using *ZIP_DEFLATED* integers `0` through `9` are accepted (see *zlib* for more information). When using *ZIP_BZIP2* integers `1` through `9` are accepted (see *bz2* for more information).

If the file is created with mode `'w'`, `'x'` or `'a'` and then *closed* without adding any files to the archive, the appropriate ZIP structures for an empty archive will be written to the file.

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

New in version 3.2: Added the ability to use *ZipFile* as a context manager.

Changed in version 3.3: Added support for *bzip2* and *lzma* compression.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Changed in version 3.5: Added support for writing to unseekable streams. Added support for the `'x'` mode.

Changed in version 3.6: Previously, a plain *RuntimeError* was raised for unrecognized compression values.

Changed in version 3.6.2: The *file* parameter accepts a *path-like object*.

Changed in version 3.7: Add the *compresslevel* parameter.

ZipFile.close()
Close the archive file. You must call *close()* before exiting your program or essential records will not be written.

ZipFile.getinfo(*name*)
Return a *ZipInfo* object with information about the archive member *name*. Calling *getinfo()* for a name not currently contained in the archive will raise a *KeyError*.

ZipFile.infolist()
Return a list containing a *ZipInfo* object for each member of the archive. The objects are in the same order as their entries in the actual ZIP file on disk if an existing archive was opened.

ZipFile.namelist()
Return a list of archive members by name.

ZipFile.open(*name*, *mode='r'*, *pwd=None*, ***, *force_zip64=False*)
Access a member of the archive as a binary file-like object. *name* can be either the name of a file within the archive or a *ZipInfo* object. The *mode* parameter, if included, must be `'r'` (the default) or `'w'`. *pwd* is the password used to decrypt encrypted ZIP files.

open() is also a context manager and therefore supports the `with` statement:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* `'r'` the file-like object (`ZipExtFile`) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, `__iter__()`, *__next__()*. These objects can operate independently of the ZipFile.

With `mode='w'`, a writable file handle is returned, which supports the *write()* method. While a writable file handle is open, attempting to read or write other files in the ZIP file will raise a *ValueError*.

When writing a file, if the file size is not known in advance but may exceed 2 GiB, pass `force_zip64=True` to ensure that the header format is capable of supporting large files. If the file size is known in advance, construct a *ZipInfo* object with *file_size* set, and use that as the *name* parameter.

Note: The *open()*, *read()* and *extract()* methods can take a filename or a *ZipInfo* object. You will appreciate this when trying to read a ZIP file that contains members with duplicate names.

Changed in version 3.6: Removed support of `mode='U'`. Use *io.TextIOWrapper* for reading compressed text files in *universal newlines* mode.

Changed in version 3.6: *open()* can now be used to write files into the archive with the `mode='w'` option.

Changed in version 3.6: Calling *open()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.extract(*member*, *path=None*, *pwd=None*)
Extract a member from the archive to the current working directory; *member* must be its full name or a *ZipInfo* object. Its file information is extracted as accurately as possible. *path* specifies a different

directory to extract to. *member* can be a filename or a *ZipInfo* object. *pwd* is the password used for encrypted files.

Returns the normalized path created (a directory or new file).

Note: If a member filename is an absolute path, a drive/UNC sharepoint and leading (back)slashes will be stripped, e.g.: `///foo/bar` becomes `foo/bar` on Unix, and `C:\foo\bar` becomes `foo\bar` on Windows. And all `".."` components in a member filename will be removed, e.g.: `../../foo../../ba..r` becomes `foo../ba..r`. On Windows illegal characters (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) replaced by underscore (`_`).

Changed in version 3.6: Calling *extract()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

Changed in version 3.6.2: The *path* parameter accepts a *path-like object*.

ZipFile.extractall(*path=None*, *members=None*, *pwd=None*)
Extract all members from the archive to the current working directory. *path* specifies a different directory to extract to. *members* is optional and must be a subset of the list returned by *namelist()*. *pwd* is the password used for encrypted files.

> **Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`. This module attempts to prevent that. See *extract()* note.

Changed in version 3.6: Calling *extractall()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

Changed in version 3.6.2: The *path* parameter accepts a *path-like object*.

ZipFile.printdir()
Print a table of contents for the archive to `sys.stdout`.

ZipFile.setpassword(*pwd*)
Set *pwd* as default password to extract encrypted files.

ZipFile.read(*name*, *pwd=None*)
Return the bytes of the file *name* in the archive. *name* is the name of the file in the archive, or a *ZipInfo* object. The archive must be open for read or append. *pwd* is the password used for encrypted files and, if specified, it will override the default password set with *setpassword()*. Calling *read()* on a ZipFile that uses a compression method other than *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* will raise a *NotImplementedError*. An error will also be raised if the corresponding compression module is not available.

Changed in version 3.6: Calling *read()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.testzip()
Read all the files in the archive and check their CRC's and file headers. Return the name of the first bad file, or else return `None`.

Changed in version 3.6: Calling *testzip()* on a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

ZipFile.write(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)
Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for

the new entry. Similarly, *compresslevel* will override the constructor if given. The archive must be open with mode `'w'`, `'x'` or `'a'`.

Note: Archive names should be relative to the archive root, that is, they should not start with a path separator.

Note: If `arcname` (or `filename`, if `arcname` is not given) contains a null byte, the name of the file in the archive will be truncated at the null byte.

Changed in version 3.6: Calling *write()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

`ZipFile.writestr`(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)

Write a file into the archive. The contents is *data*, which may be either a *str* or a *bytes* instance; if it is a *str*, it is encoded as UTF-8 first. *zinfo_or_arcname* is either the file name it will be given in the archive, or a *ZipInfo* instance. If it's an instance, at least the filename, date, and time must be given. If it's a name, the date and time is set to the current date and time. The archive must be opened with mode `'w'`, `'x'` or `'a'`.

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance). Similarly, *compresslevel* will override the constructor if given.

Note: When passing a *ZipInfo* instance as the *zinfo_or_arcname* parameter, the compression method used will be that specified in the *compress_type* member of the given *ZipInfo* instance. By default, the *ZipInfo* constructor sets this member to *ZIP_STORED*.

Changed in version 3.2: The *compress_type* argument.

Changed in version 3.6: Calling *writestr()* on a ZipFile created with mode `'r'` or a closed ZipFile will raise a *ValueError*. Previously, a *RuntimeError* was raised.

The following data attributes are also available:

`ZipFile.filename`

Name of the ZIP file.

`ZipFile.debug`

The level of debug output to use. This may be set from `0` (the default, no output) to `3` (the most output). Debugging information is written to `sys.stdout`.

`ZipFile.comment`

The comment associated with the ZIP file as a *bytes* object. If assigning a comment to a *ZipFile* instance created with mode `'w'`, `'x'` or `'a'`, it should be no longer than 65535 bytes. Comments longer than this will be truncated.

13.5.2 PyZipFile Objects

The *PyZipFile* constructor takes the same parameters as the *ZipFile* constructor, and one additional parameter, *optimize*.

`class zipfile.PyZipFile`(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)

New in version 3.2: The *optimize* parameter.

Changed in version 3.4: ZIP64 extensions are enabled by default.

Instances have one method in addition to those of *ZipFile* objects:

writepy(*pathname*, *basename=''*, *filterfunc=None*)

Search for files `*.py` and add the corresponding file to the archive.

If the *optimize* parameter to *PyZipFile* was not given or `-1`, the corresponding file is a `*.pyc` file, compiling if necessary.

If the *optimize* parameter to *PyZipFile* was `0`, `1` or `2`, only files with that optimization level (see *compile()*) are added to the archive, compiling if necessary.

If *pathname* is a file, the filename must end with `.py`, and just the (corresponding `*.pyc`) file is added at the top level (no path information). If *pathname* is a file that does not end with `.py`, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.pyc` are added at the top level. If the directory is a package directory, then all `*.pyc` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively in sorted order.

basename is intended for internal use only.

filterfunc, if given, must be a function taking a single string argument. It will be passed each path (including each individual full file path) before it is added to the archive. If *filterfunc* returns a false value, the path will not be added, and if it is a directory its contents will be ignored. For example, if our test files are all either in `test` directories or start with the string `test_`, we can use a *filterfunc* to exclude them:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

The *writepy()* method makes archives with file names like this:

```
string.pyc                    # Top level name
test/__init__.pyc             # Package directory
test/testall.pyc              # Module test.testall
test/bogus/__init__.pyc       # Subpackage directory
test/bogus/myfile.pyc         # Submodule test.bogus.myfile
```

New in version 3.4: The *filterfunc* parameter.

Changed in version 3.6.2: The *pathname* parameter accepts a *path-like object*.

Changed in version 3.7: Recursion sorts directory entries.

13.5.3 ZipInfo Objects

Instances of the *ZipInfo* class are returned by the *getinfo()* and *infolist()* methods of *ZipFile* objects. Each object stores information about a single member of the ZIP archive.

There is one classmethod to make a *ZipInfo* instance for a filesystem file:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*)

Construct a *ZipInfo* instance for a file on the filesystem, in preparation for adding it to a zip file.

filename should be the path to a file or directory on the filesystem.

If *arcname* is specified, it is used as the name within the archive. If *arcname* is not specified, the name will be the same as *filename*, but with any drive letter and leading path separators removed.

New in version 3.6.

Changed in version 3.6.2: The *filename* parameter accepts a *path-like object*.

Instances have the following methods and attributes:

ZipInfo.is_dir()
Return `True` if this archive member is a directory.

This uses the entry's name: directories should always end with `/`.

New in version 3.6.

ZipInfo.filename
Name of the file in the archive.

ZipInfo.date_time
The time and date of the last modification to the archive member. This is a tuple of six values:

Index	Value
0	Year (>= 1980)
1	Month (one-based)
2	Day of month (one-based)
3	Hours (zero-based)
4	Minutes (zero-based)
5	Seconds (zero-based)

Note: The ZIP file format does not support timestamps before 1980.

ZipInfo.compress_type
Type of compression for the archive member.

ZipInfo.comment
Comment for the individual archive member as a *bytes* object.

ZipInfo.extra
Expansion field data. The PKZIP Application Note contains some comments on the internal structure of the data contained in this *bytes* object.

ZipInfo.create_system
System which created ZIP archive.

ZipInfo.create_version
PKZIP version which created ZIP archive.

ZipInfo.extract_version
PKZIP version needed to extract archive.

ZipInfo.reserved
Must be zero.

ZipInfo.flag_bits
ZIP flag bits.

ZipInfo.volume
Volume number of file header.

ZipInfo.internal_attr
Internal attributes.

ZipInfo.external_attr
External file attributes.

ZipInfo.header_offset
Byte offset to the file header.

ZipInfo.CRC
CRC-32 of the uncompressed file.

ZipInfo.compress_size
Size of the compressed data.

ZipInfo.file_size
Size of the uncompressed file.

13.5.4 Command-Line Interface

The *zipfile* module provides a simple command-line interface to interact with ZIP archives.

If you want to create a new ZIP archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

If you want to extract a ZIP archive into the specified directory, use the `-e` option:

```
$ python -m zipfile -e monty.zip target-dir/
```

For a list of the files in a ZIP archive, use the `-l` option:

```
$ python -m zipfile -l monty.zip
```

Command-line options

-l <zipfile>
--list <zipfile>
List files in a zipfile.

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
Create zipfile from source files.

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
Extract zipfile into target directory.

-t <zipfile>
--test <zipfile>
Test whether the zipfile is valid or not.

13.6 tarfile — Read and write tar archive files

Source code: Lib/tarfile.py

The *tarfile* module makes it possible to read and write tar archives, including those using gzip, bz2 and lzma compression. Use the *zipfile* module to read or write `.zip` files, or the higher-level functions in *shutil*.

Some facts and figures:

- reads and writes *gzip*, *bz2* and *lzma* compressed archives if the respective modules are available.
- read/write support for the POSIX.1-1988 (ustar) format.
- read/write support for the GNU tar format including *longname* and *longlink* extensions, read-only support for all variants of the *sparse* extension including restoration of sparse files.
- read/write support for the POSIX.1-2001 (pax) format.
- handles directories, regular files, hardlinks, symbolic links, fifos, character devices and block devices and is able to acquire and restore file information like timestamp, access permissions and owner.

Changed in version 3.3: Added support for *lzma* compression.

`tarfile.open`(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

Return a *TarFile* object for the pathname *name*. For detailed information on *TarFile* objects and the keyword arguments that are allowed, see *TarFile Objects*.

mode has to be a string of the form `'filemode[:compression]'`, it defaults to `'r'`. Here is a full list of mode combinations:

mode	action
`'r'` or `'r:*'`	Open for reading with transparent compression (recommended).
`'r:'`	Open for reading exclusively without compression.
`'r:gz'`	Open for reading with gzip compression.
`'r:bz2'`	Open for reading with bzip2 compression.
`'r:xz'`	Open for reading with lzma compression.
`'x'` or `'x:'`	Create a tarfile exclusively without compression. Raise an *FileExistsError* exception if it already exists.
`'x:gz'`	Create a tarfile with gzip compression. Raise an *FileExistsError* exception if it already exists.
`'x:bz2'`	Create a tarfile with bzip2 compression. Raise an *FileExistsError* exception if it already exists.
`'x:xz'`	Create a tarfile with lzma compression. Raise an *FileExistsError* exception if it already exists.
`'a'` or `'a:'`	Open for appending with no compression. The file is created if it does not exist.
`'w'` or `'w:'`	Open for uncompressed writing.
`'w:gz'`	Open for gzip compressed writing.
`'w:bz2'`	Open for bzip2 compressed writing.
`'w:xz'`	Open for lzma compressed writing.

Note that `'a:gz'`, `'a:bz2'` or `'a:xz'` is not possible. If *mode* is not suitable to open a certain (compressed) file for reading, *ReadError* is raised. Use *mode* `'r'` to avoid this. If a compression method is not supported, *CompressionError* is raised.

If *fileobj* is specified, it is used as an alternative to a *file object* opened in binary mode for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'`, *tarfile.open()* accepts the keyword argument *compresslevel* (default `9`) to specify the compression level of the file.

For special purposes, there is a second format for *mode*: `'filemode|[compression]'`. *tarfile.open()* will return a *TarFile* object that processes its data as a stream of blocks. No random seeking will be done on the file. If given, *fileobj* may be any object that has a `read()` or `write()` method (depending on the *mode*). *bufsize* specifies the blocksize and defaults to `20 * 512` bytes. Use this variant in combination with e.g. `sys.stdin`, a socket *file object* or a tape device. However, such a *TarFile* object is limited in that it does not allow random access, see *Examples*. The currently possible modes:

Mode	Action
`'r|*'`	Open a *stream* of tar blocks for reading with transparent compression.
`'r|'`	Open a *stream* of uncompressed tar blocks for reading.
`'r|gz'`	Open a gzip compressed *stream* for reading.
`'r|bz2'`	Open a bzip2 compressed *stream* for reading.
`'r|xz'`	Open an lzma compressed *stream* for reading.
`'w|'`	Open an uncompressed *stream* for writing.
`'w|gz'`	Open a gzip compressed *stream* for writing.
`'w|bz2'`	Open a bzip2 compressed *stream* for writing.
`'w|xz'`	Open an lzma compressed *stream* for writing.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

class tarfile.TarFile
Class for reading and writing tar archives. Do not use this class directly: use *tarfile.open()* instead. See *TarFile Objects*.

tarfile.is_tarfile(*name*)
Return *True* if *name* is a tar archive file, that the *tarfile* module can read.

The *tarfile* module defines the following exceptions:

exception tarfile.TarError
Base class for all *tarfile* exceptions.

exception tarfile.ReadError
Is raised when a tar archive is opened, that either cannot be handled by the *tarfile* module or is somehow invalid.

exception tarfile.CompressionError
Is raised when a compression method is not supported or when the data cannot be decoded properly.

exception tarfile.StreamError
Is raised for the limitations that are typical for stream-like *TarFile* objects.

exception tarfile.ExtractError
Is raised for *non-fatal* errors when using *TarFile.extract()*, but only if `TarFile.errorlevel== 2`.

exception tarfile.HeaderError
Is raised by *TarInfo.frombuf()* if the buffer it gets is invalid.

The following constants are available at the module level:

tarfile.ENCODING
The default character encoding: `'utf-8'` on Windows, the value returned by *sys.getfilesystemencoding()* otherwise.

Each of the following constants defines a tar archive format that the *tarfile* module is able to create. See section *Supported tar formats* for details.

tarfile.USTAR_FORMAT
POSIX.1-1988 (ustar) format.

`tarfile.GNU_FORMAT`
GNU tar format.

`tarfile.PAX_FORMAT`
POSIX.1-2001 (pax) format.

`tarfile.DEFAULT_FORMAT`
The default format for creating archives. This is currently *GNU_FORMAT*.

See also:

Module *zipfile* Documentation of the *zipfile* standard module.

Archiving operations Documentation of the higher-level archiving facilities provided by the standard *shutil* module.

GNU tar manual, Basic Tar Format Documentation for tar archive files, including GNU tar extensions.

13.6.1 TarFile Objects

The *TarFile* object provides an interface to a tar archive. A tar archive is a sequence of blocks. An archive member (a stored file) is made up of a header block followed by data blocks. It is possible to store a file in a tar archive several times. Each archive member is represented by a *TarInfo* object, see *TarInfo Objects* for details.

A *TarFile* object can be used as a context manager in a `with` statement. It will automatically be closed when the block is completed. Please note that in the event of an exception an archive opened for writing will not be finalized; only the internally used file object will be closed. See the *Examples* section for a use case.

New in version 3.2: Added support for the context management protocol.

`class tarfile.TarFile(`*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=0*`)`

All following arguments are optional and can be accessed as instance attributes as well.

name is the pathname of the archive. *name* may be a *path-like object*. It can be omitted if *fileobj* is given. In this case, the file object's `name` attribute is used if it exists.

mode is either `'r'` to read from an existing archive, `'a'` to append data to an existing file, `'w'` to create a new file overwriting an existing one, or `'x'` to create a new file only if it does not already exist.

If *fileobj* is given, it is used for reading or writing data. If it can be determined, *mode* is overridden by *fileobj*'s mode. *fileobj* will be used from position 0.

Note: *fileobj* is not closed, when *TarFile* is closed.

---

format controls the archive format. It must be one of the constants *USTAR_FORMAT*, *GNU_FORMAT* or *PAX_FORMAT* that are defined at module level.

The *tarinfo* argument can be used to replace the default *TarInfo* class with a different one.

If *dereference* is *False*, add symbolic and hard links to the archive. If it is *True*, add the content of the target files to the archive. This has no effect on systems that do not support symbolic links.

If *ignore_zeros* is *False*, treat an empty block as the end of the archive. If it is *True*, skip empty (and invalid) blocks and try to get as many members as possible. This is only useful for reading concatenated or damaged archives.

debug can be set from `0` (no debug messages) up to `3` (all debug messages). The messages are written to `sys.stderr`.

If *errorlevel* is `0`, all errors are ignored when using *TarFile.extract()*. Nevertheless, they appear as error messages in the debug output, when debugging is enabled. If `1`, all *fatal* errors are raised as *OSError* exceptions. If `2`, all *non-fatal* errors are raised as *TarError* exceptions as well.

The *encoding* and *errors* arguments define the character encoding to be used for reading or writing the archive and how conversion errors are going to be handled. The default settings will work for most users. See section *Unicode issues* for in-depth information.

The *pax_headers* argument is an optional dictionary of strings which will be added as a pax global header if *format* is *PAX_FORMAT*.

Changed in version 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

Changed in version 3.5: The `'x'` (exclusive creation) mode was added.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

classmethod TarFile.**open**(...)
Alternative constructor. The *tarfile.open()* function is actually a shortcut to this classmethod.

TarFile.**getmember**(*name*)
Return a *TarInfo* object for member *name*. If *name* can not be found in the archive, *KeyError* is raised.

Note: If a member occurs more than once in the archive, its last occurrence is assumed to be the most up-to-date version.

TarFile.**getmembers**()
Return the members of the archive as a list of *TarInfo* objects. The list has the same order as the members in the archive.

TarFile.**getnames**()
Return the members as a list of their names. It has the same order as the list returned by *getmembers()*.

TarFile.**list**(*verbose=True*, ***, *members=None*)
Print a table of contents to `sys.stdout`. If *verbose* is *False*, only the names of the members are printed. If it is *True*, output similar to that of `ls -l` is produced. If optional *members* is given, it must be a subset of the list returned by *getmembers()*.

Changed in version 3.5: Added the *members* parameter.

TarFile.**next**()
Return the next member of the archive as a *TarInfo* object, when *TarFile* is opened for reading. Return *None* if there is no more available.

TarFile.**extractall**(*path="."*, *members=None*, ***, *numeric_owner=False*)
Extract all members from the archive to the current working directory or directory *path*. If optional *members* is given, it must be a subset of the list returned by *getmembers()*. Directory information like owner, modification time and permissions are set after all members have been extracted. This is done to work around two problems: A directory's modification time is reset each time a file is created in it. And, if a directory's permissions do not allow writing, extracting files to it will fail.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

> **Warning:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of *path*, e.g. members that have absolute filenames starting with `"/"` or filenames with two dots `".."`.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a *path-like object*.

`TarFile.extract`(*member*, *path=""*, *set_attrs=True*, ***, *numeric_owner=False*)
Extract a member from the archive to the current working directory, using its full name. Its file information is extracted as accurately as possible. *member* may be a filename or a *TarInfo* object. You can specify a different directory using *path*. *path* may be a *path-like object*. File attributes (owner, mtime, mode) are set unless *set_attrs* is false.

If *numeric_owner* is *True*, the uid and gid numbers from the tarfile are used to set the owner/group for the extracted files. Otherwise, the named values from the tarfile are used.

Note: The *extract()* method does not take care of several extraction issues. In most cases you should consider using the *extractall()* method.

> **Warning:** See the warning for *extractall()*.

Changed in version 3.2: Added the *set_attrs* parameter.

Changed in version 3.5: Added the *numeric_owner* parameter.

Changed in version 3.6: The *path* parameter accepts a *path-like object*.

`TarFile.extractfile`(*member*)
Extract a member from the archive as a file object. *member* may be a filename or a *TarInfo* object. If *member* is a regular file or a link, an *io.BufferedReader* object is returned. Otherwise, *None* is returned.

Changed in version 3.3: Return an *io.BufferedReader* object.

`TarFile.add`(*name*, *arcname=None*, *recursive=True*, ***, *filter=None*)
Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to *False*. Recursion adds entries in sorted order. If *filter* is given, it should be a function that takes a *TarInfo* object argument and returns the changed *TarInfo* object. If it instead returns *None* the *TarInfo* object will be excluded from the archive. See *Examples* for an example.

Changed in version 3.2: Added the *filter* parameter.

Changed in version 3.7: Recursion adds entries in sorted order.

`TarFile.addfile`(*tarinfo*, *fileobj=None*)
Add the *TarInfo* object *tarinfo* to the archive. If *fileobj* is given, it should be a *binary file*, and `tarinfo.size` bytes are read from it and added to the archive. You can create *TarInfo* objects directly, or by using *gettarinfo()*.

`TarFile.gettarinfo`(*name=None*, *arcname=None*, *fileobj=None*)
Create a *TarInfo* object from the result of *os.stat()* or equivalent on an existing file. The file is either named by *name*, or specified as a *file object* *fileobj* with a file descriptor. *name* may be a *path-like object*. If given, *arcname* specifies an alternative name for the file in the archive, otherwise, the name is taken from *fileobj*'s *name* attribute, or the *name* argument. The name should be a text string.

You can modify some of the *TarInfo*'s attributes before you add it using *addfile()*. If the file object is not an ordinary file object positioned at the beginning of the file, attributes such as *size* may need modifying. This is the case for objects such as *GzipFile*. The *name* may also be modified, in which case *arcname* could be a dummy string.

Changed in version 3.6: The *name* parameter accepts a *path-like object*.

`TarFile.close()`
Close the *TarFile*. In write mode, two finishing zero blocks are appended to the archive.

`TarFile.pax_headers`
A dictionary containing key-value pairs of pax global headers.

13.6.2 TarInfo Objects

A *TarInfo* object represents one member in a *TarFile*. Aside from storing all required attributes of a file (like file type, size, time, permissions, owner etc.), it provides some useful methods to determine its type. It does *not* contain the file's data itself.

TarInfo objects are returned by *TarFile*'s methods `getmember()`, `getmembers()` and `gettarinfo()`.

`class tarfile.TarInfo(`*name=""*`)`
Create a *TarInfo* object.

`classmethod TarInfo.frombuf(`*buf*, *encoding*, *errors*`)`
Create and return a *TarInfo* object from string buffer *buf*.

Raises *HeaderError* if the buffer is invalid.

`classmethod TarInfo.fromtarfile(`*tarfile*`)`
Read the next member from the *TarFile* object *tarfile* and return it as a *TarInfo* object.

`TarInfo.tobuf(`*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*`)`
Create a string buffer from a *TarInfo* object. For information on the arguments see the constructor of the *TarFile* class.

Changed in version 3.2: Use `'surrogateescape'` as the default for the *errors* argument.

A `TarInfo` object has the following public data attributes:

`TarInfo.name`
Name of the archive member.

`TarInfo.size`
Size in bytes.

`TarInfo.mtime`
Time of last modification.

`TarInfo.mode`
Permission bits.

`TarInfo.type`
File type. *type* is usually one of these constants: `REGTYPE`, `AREGTYPE`, `LNKTYPE`, `SYMTYPE`, `DIRTYPE`, `FIFOTYPE`, `CONTTYPE`, `CHRTYPE`, `BLKTYPE`, `GNUTYPE_SPARSE`. To determine the type of a *TarInfo* object more conveniently, use the `is*()` methods below.

`TarInfo.linkname`
Name of the target file name, which is only present in *TarInfo* objects of type `LNKTYPE` and `SYMTYPE`.

`TarInfo.uid`
User ID of the user who originally stored this member.

TarInfo.gid
Group ID of the user who originally stored this member.

TarInfo.uname
User name.

TarInfo.gname
Group name.

TarInfo.pax_headers
A dictionary containing key-value pairs of an associated pax extended header.

A *TarInfo* object also provides some convenient query methods:

TarInfo.isfile()
Return *True* if the Tarinfo object is a regular file.

TarInfo.isreg()
Same as *isfile()*.

TarInfo.isdir()
Return *True* if it is a directory.

TarInfo.issym()
Return *True* if it is a symbolic link.

TarInfo.islnk()
Return *True* if it is a hard link.

TarInfo.ischr()
Return *True* if it is a character device.

TarInfo.isblk()
Return *True* if it is a block device.

TarInfo.isfifo()
Return *True* if it is a FIFO.

TarInfo.isdev()
Return *True* if it is one of character device, block device or FIFO.

13.6.3 Command-Line Interface

New in version 3.4.

The *tarfile* module provides a simple command-line interface to interact with tar archives.

If you want to create a new tar archive, specify its name after the `-c` option and then list the filename(s) that should be included:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

Passing a directory is also acceptable:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

If you want to extract a tar archive into the current directory, use the `-e` option:

```
$ python -m tarfile -e monty.tar
```

You can also extract a tar archive into a different directory by passing the directory's name:

```
$ python -m tarfile -e monty.tar  other-dir/
```

For a list of the files in a tar archive, use the *-l* option:

```
$ python -m tarfile -l monty.tar
```

Command-line options

-l <tarfile>
--list <tarfile>
List files in a tarfile.

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
Create tarfile from source files.

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
Extract tarfile into the current directory if *output_dir* is not specified.

-t <tarfile>
--test <tarfile>
Test whether the tarfile is valid or not.

-v, --verbose
Verbose output.

13.6.4 Examples

How to extract an entire tar archive to the current working directory:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

How to extract a subset of a tar archive with *TarFile.extractall()* using a generator function instead of a list:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

How to create an uncompressed tar archive from a list of filenames:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

The same example using the `with` statement:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

How to read a gzip compressed tar archive and display some member information:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

How to create an archive and reset the user information using the *filter* parameter in *TarFile.add()*:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5 Supported tar formats

There are three tar formats that can be created with the *tarfile* module:

- The POSIX.1-1988 ustar format (*USTAR_FORMAT*). It supports filenames up to a length of at best 256 characters and linknames up to 100 characters. The maximum file size is 8 GiB. This is an old and limited but widely supported format.
- The GNU tar format (*GNU_FORMAT*). It supports long filenames and linknames, files bigger than 8 GiB and sparse files. It is the de facto standard on GNU/Linux systems. *tarfile* fully supports the GNU tar extensions for long names, sparse file support is read-only.
- The POSIX.1-2001 pax format (*PAX_FORMAT*). It is the most flexible format with virtually no limits. It supports long filenames and linknames, large files and stores pathnames in a portable way. However, not all tar implementations today are able to handle pax archives properly.

  The *pax* format is an extension to the existing *ustar* format. It uses extra headers for information that cannot be stored otherwise. There are two flavours of pax headers: Extended headers only affect the

subsequent file header, global headers are valid for the complete archive and affect all following files. All the data in a pax header is encoded in *UTF-8* for portability reasons.

There are some more variants of the tar format which can be read, but not created:

- The ancient V7 format. This is the first tar format from Unix Seventh Edition, storing only regular files and directories. Names must not be longer than 100 characters, there is no user/group name information. Some archives have miscalculated header checksums in case of fields with non-ASCII characters.
- The SunOS tar extended format. This format is a variant of the POSIX.1-2001 pax format, but is not compatible.

13.6.6 Unicode issues

The tar format was originally conceived to make backups on tape drives with the main focus on preserving file system information. Nowadays tar archives are commonly used for file distribution and exchanging archives over networks. One problem of the original format (which is the basis of all other formats) is that there is no concept of supporting different character encodings. For example, an ordinary tar archive created on a *UTF-8* system cannot be read correctly on a *Latin-1* system if it contains non-*ASCII* characters. Textual metadata (like filenames, linknames, user/group names) will appear damaged. Unfortunately, there is no way to autodetect the encoding of an archive. The pax format was designed to solve this problem. It stores non-ASCII metadata using the universal character encoding *UTF-8*.

The details of character conversion in *tarfile* are controlled by the *encoding* and *errors* keyword arguments of the *TarFile* class.

encoding defines the character encoding to use for the metadata in the archive. The default value is *sys.getfilesystemencoding()* or `'ascii'` as a fallback. Depending on whether the archive is read or written, the metadata must be either decoded or encoded. If *encoding* is not set appropriately, this conversion may fail.

The *errors* argument defines how characters are treated that cannot be converted. Possible values are listed in section *Error Handlers*. The default scheme is `'surrogateescape'` which Python also uses for its file system calls, see *File Names, Command Line Arguments, and Environment Variables*.

In case of *PAX_FORMAT* archives, *encoding* is generally not needed because all the metadata is stored using *UTF-8*. *encoding* is only used in the rare cases when binary pax headers are decoded or when strings with surrogate characters are stored.