12.5. `tarfile` — Read and write tar archive files

New in version 2.3.

Source code: Lib/tarfile.py

The `tarfile` module makes it possible to read and write tar archives, including those using gzip or bz2 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` and `bz2` 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 the *sparse* extension.
- read/write support for the POSIX.1-2001 (pax) format.

  New in version 2.6.
- 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.

`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.
`'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.

Note that `'a:gz'` or `'a:bz2'` 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 for *name*. It is supposed to be at position 0.

For modes `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r: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.
`'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.

class `tarfile.`**`TarFile`**

Class for reading and writing tar archives. Do not use this class directly, better 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.

class `tarfile.`**`TarFileCompat`**(*filename*, *mode='r'*, *compression=TAR_PLAIN*)

Class for limited access to tar archives with a `zipfile`-like interface. Please consult the documentation of the `zipfile` module for more details. *compression* must be one of the following constants:

`TAR_PLAIN`

Constant for an uncompressed tar archive.

`TAR_GZIPPED`

Constant for a `gzip` compressed tar archive.

Deprecated since version 2.6: The `TarFileCompat` class has been removed in Python 3.

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`.

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.

exception `tarfile.`**`HeaderError`**

Is raised by `TarInfo.frombuf()` if the buffer it gets is invalid.

New in version 2.6.

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.

12.5.1. 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 2.7: 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=None*, *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. 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 or `'w'` to create a new file overwriting an existing one.

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.

New in version 2.6.

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

New in version 2.6.

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` or `IOError` exceptions. If `2`, all *non-fatal* errors are raised as `TarError` exceptions as well.

The *encoding* and *errors* arguments control the way strings are converted to unicode objects and vice versa. The default settings will work for most users. See section Unicode issues for in-depth information.

New in version 2.6.

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

New in version 2.6.

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*)

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.

`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*)

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.

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 `".."`.

New in version 2.5.

`TarFile.`**`extract`**(*member*, *path=""*)

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*.

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()`.

`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, a file-like object is returned. If *member* is a link, a file-like object is constructed from the link's target. If *member* is none of the above, `None` is returned.

Note: The file-like object is read-only. It provides the methods `read()`, `readline()`, `readlines()`, `seek()`, `tell()`, and `close()`, and also supports iteration over its lines.

`TarFile.`**`add`**(*name*, *arcname=None*, *recursive=True*, *exclude=None*, *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`. If *exclude* is given it must be a function that takes one filename argument and returns a boolean value. Depending on this value the respective file is either excluded (`True`) or added (`False`). If *filter* is specified it must 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 2.6: Added the *exclude* parameter.

Changed in version 2.7: Added the *filter* parameter.

Deprecated since version 2.7: The *exclude* parameter is deprecated, please use the *filter* parameter instead. For maximum portability, *filter* should be used as a keyword argument rather than as a positional argument so that code won't be affected when *exclude* is ultimately removed.

`TarFile.`**`addfile`**(*tarinfo*, *fileobj=None*)

Add the `TarInfo` object *tarinfo* to the archive. If *fileobj* is given, `tarinfo.size` bytes are read from it and added to the archive. You can create `TarInfo` objects directly, or by using `gettarinfo()`.

Note: On Windows platforms, *fileobj* should always be opened with mode `'rb'` to avoid irritation about the file size.

`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. 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.

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.

`TarFile.`**`close`**()

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

`TarFile.`**`posix`**

Setting this to `True` is equivalent to setting the `format` attribute to `USTAR_FORMAT`, `False` is equivalent to `GNU_FORMAT`.

Changed in version 2.4: posix defaults to `False`.

Deprecated since version 2.6: Use the `format` attribute instead.

`TarFile.`**`pax_headers`**

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

New in version 2.6.

12.5.2. 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.

`TarInfo.`**`frombuf`**(*buf*)

Create and return a `TarInfo` object from string buffer *buf*.

New in version 2.6: Raises `HeaderError` if the buffer is invalid..

`TarInfo.`**`fromtarfile`**(*tarfile*)

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

New in version 2.6.

`TarInfo.`**`tobuf`**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='strict'*)

Create a string buffer from a `TarInfo` object. For information on the arguments see the constructor of the `TarFile` class.

Changed in version 2.6: The arguments were added.

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.

New in version 2.6.

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.

12.5.3. 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",
    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()
```

12.5.4. 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 gigabytes. 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 gigabytes 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.

12.5.5. 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 (that all other formats are merely variants of) 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. Names (i.e. filenames, linknames, user/group names) containing these characters 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 names using the universal character encoding *UTF-8*. When a pax archive is read, these *UTF-8* names are converted to the encoding of the local file system.

The details of unicode conversion are controlled by the *encoding* and *errors* keyword arguments of the `TarFile` class.

The default value for *encoding* is the local character encoding. It is deduced from `sys.getfilesystemencoding()` and `sys.getdefaultencoding()`. In read mode, *encoding* is used exclusively to convert unicode names from a pax archive to strings in the local character encoding. In write mode, the use of *encoding* depends on the chosen archive format. In case of `PAX_FORMAT`, input names that contain non-ASCII characters need to be decoded before being stored as *UTF-8* strings. The other formats do not make use of *encoding* unless unicode objects are used as input names. These are converted to 8-bit character strings before they are added to the archive.

The *errors* argument defines how characters are treated that cannot be converted to or from *encoding*. Possible values are listed in section Codec Base Classes. In read mode, there is an additional scheme `'utf-8'` which means that bad characters are replaced by their *UTF-8* representation. This is the default scheme. In write mode the default value for *errors* is `'strict'` to ensure that name information is not altered unnoticed.