
compressor Documentation

Release 0.2.0

Mariano Anaya

May 22, 2022

Contents

1	Using the Application	3
1.1	Basic usage	3
2	compressor	5
2.1	compressor package	5
3	Indices and tables	11
	Python Module Index	13
	Index	15

Pycompressor is a tool for compressing text files into smaller ones, as well as extracting compressed files back into the original content.

For example, in order to compress one file:

```
$ pycompress -c /usr/share/dict/words -d /tmp/compressed.zf
```

The original file, in this example has a size of ~4 . 8M, and the tool left the resulting file at /tmp/compressed.zf, with a size of ~2 . 7M.

In order to extract it:

```
$ pycompress -x /tmp/compressed.zf -d /tmp/original
```

You can specify the name of the resulting file with the `-d` flag. If you don't indicate a name for the resulting file, the default will be `<original-file>.comp`.

For the full options, run:

```
$ pycompress -h
```

Contents:

Using the Application

This section explains how the application is used from the command line interface (`cli`), detailing which parameters are accepted and how they work.

1.1 Basic usage

1.1.1 Compressing a File

You can start using the program by just running it, and telling `pycompressor` the name of the file you'd like to compress, for example:

```
$ pycompress -c /usr/share/dict/words
```

The `-c` parameter stands for “compress”, and if nothing else is specified, the resulting file will be left on the current directory, with the base name of the provided file and the `.comp` suffix. In this example, the result of will be a file named `words.comp`.

You can change the name of the resulting file, by passing the `-d` (destination) flag, like in:

```
$ pycompress -c /usr/share/dict/words -d /tmp/compressed.zf
```

In this case the resulting file (after compressed) will be `/tmp/compressed.zf`.

1.1.2 Extracting a file

If you want to recover the original file from a binary, compressed one, use the `-x` (extract) flag:

```
$ pycompress -x /tmp/compressed.zf
```

If a name for the resulting file is not specified, it'll assume the base name provided with the `.extr` suffix, in the local path of where the command is being applied. In this case, it would be `compressed.zf.extr`.

You can also indicate the name of the destination file, again with the `-d` parameter:

```
$ pycompress -x /tmp/compressed.zf -d /tmp/original
```

The destination file in this case, indicates that after extracted the file is written in `/tmp/original`.

2.1 compressor package

2.1.1 Submodules

2.1.2 Module contents

compressor entry point

2.1.3 lib module

compressor.lib

High-level functions exposed as a library, that can be imported.

`compressor.lib.compress_file(filename: str, dest_file: str = "") → None`

Open the <filename> and compress its contents on a new one.

Parameters

- **filename** (str) – The path to the source file to compress.
- **dest_file** (str) – The name of the target file. If not provided (None), a default will be used with <filename>.comp

`compressor.lib.open_text_file()`

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes

append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use `newline` to control universal newlines mode.

`buffering` is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on `io.DEFAULT_BUFFER_SIZE`. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which `isatty()` returns True) use line buffering. Other text files use the policy described above for binary files.

`encoding` is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the `codecs` module for the list of supported encodings.

`errors` is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a *ValueError* exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for `codecs.register` or run `'help(codecs.Codec)'` for a list of the permitted encoding error strings.

`newline` controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if `newline` is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslating. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslating.

- On output, if `newline` is `None`, any ‘n’ characters written are translated to the system default line separator, `os.linesep`. If `newline` is ‘’ or ‘n’, no translation takes place. If `newline` is any of the other legal values, any ‘n’ characters written are translated to the given string.

If `closefd` is `False`, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be `True` in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing `os.open` as *opener* results in functionality similar to passing `None`).

`open()` returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When `open()` is used to open a file in a text mode (‘w’, ‘r’, ‘wt’, ‘rt’, etc.), it returns a `TextIOWrapper`. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a `BufferedReader`; in write binary and append binary modes, it returns a `BufferedWriter`, and in read/write mode, it returns a `BufferedRandom`.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings `StringIO` can be used like a file opened in a text mode, and for bytes a `BytesIO` can be used like a file opened in a binary mode.

2.1.4 cli module

Compressor CLI (command-line interface) module. Exposes the entry point to the program for executing as command line.

`compressor.cli.argument_parser()` → `argparse.ArgumentParser`

Create the argument parser object to be used for parsing the arguments from `sys.argv`

`compressor.cli.main()` → `int`

Program cli

Returns Status code of the program.

Return type `int`

`compressor.cli.main_engine(filename: str, extract: bool = False, compress: bool = True, dest_file=None)` → `int`

Main functionality for the program cli or call as library. *extract* & *compress* must have opposite values.

Return type `int`

Parameters

- **filename** (`str`) – Path to the source file to process.
- **extract** (`bool`) – If `True`, sets the program for a extraction.
- **compress** (`bool`) – If `True`, the program should compress a file.
- **dest_file** – Optional name of the target file.

Returns 0 if executed without problems.

`compressor.cli.parse_arguments(args=None)` → `dict`

Parse the command-line (cli) provided arguments, and return a mapping of the options selected by the user with their values.

Returns `dict` with the kwargs provided in cli

2.1.5 compressor.core module

`compressor.core`

Low-level functionality with the core of the process that the main program makes use of.

It contains auxiliary functions.

`compressor.core.compress_and_save_content` (*input_filename: str, output_file: io, table: dict*)
→ None

Opens and processes <input_filename>. Iterates over the file and writes the contents on output_file.

Parameters

- **input_filename** (*str*) – the source to be compressed
- **output_file** (*io*) – opened file where to write the outcome
- **table** (*dict*) – mapping table for the char encoding

`compressor.core.create_tree_code` (*charset: List[compressor.char_node.CharNode]*) → *compressor.char_node.CharNode*

Receives a :list: of :CharNode: (characters) charset, namely leaves in the tree, and returns a tree with the corresponding prefix-free code.

Return type CharNode

Parameters **charset** – iterable with all the characters to process.

Returns iterable with a tree of the prefix-free code for the charset.

`compressor.core.decode_file_content` (*compfile: io, table: dict, checksum: int*) → *str*

Reconstruct the remaining part of the <compfile>, starting right after the metadata, decoding each bit according to the <table>.

`compressor.core.open_text_file` ()

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless `closefd` is set to `False`.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: `locale.getpreferredencoding(False)` is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which *isatty()* returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a *ValueError* exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for *codecs.register* or run '*help(codecs.Codec)*' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any 'n' characters written are translated to the system default line separator, *os.linesep*. If newline is '' or 'n', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.

If *closefd* is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing *os.open* as *opener* results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When *open()* is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a *TextIOWrapper*. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a *BufferedReader*; in write binary and append binary modes, it returns a *BufferedWriter*, and in read/write mode, it returns a *BufferedRandom*.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings *StringIO* can be used like a file opened in a text mode, and for bytes a *BytesIO* can be used like a file opened in a binary mode.

`compressor.core.parse_tree_code` (*tree*: `compressor.char_node.CharNode`, *table*: `dict = None`,
code: `bytes = b''`) → `dict`

Given the tree with the chars-frequency processed, return a table that maps each character with its binary representation on the new code:

left → 0

right → 1

Return type `dict`

Parameters

- **tree** (`CharNode`) – iterable with the tree as returned by `create_tree_code`
- **table** (`dict`) – Map with the translation for the characters to its code in the new system (prefix-free).
- **code** (`bytes`) – The code prefix so far.

Returns Mapping with with the original char to its new code.

`compressor.core.process_frequencies` (*stream*: `Sequence[str]` →
`List[compressor.char_node.CharNode]`)

Given a stream of text, return a list of `CharNode` with the frequencies for each character.

Parameters **stream** – sequence with all the characters.

`compressor.core.process_line_compression` (*buffer_line*: `str`, *output_file*: `io`, *table*: `dict`) →
`None`

Transform *buffer_line* into the new code, per-byte, based on *table* and save the new byte-stream into *output_file*.

Parameters

- **buffer_line** (`str`) – a chunk of the text to process.
- **output_file** (`io`) – The opened file where to write the result.
- **table** (`dict`) – Translation table for the characters in *buffer_line*.

`compressor.core.retrieve_compressed_file` (*filename*: `str`, *dest_file*: `str = ''`) → `None`
EXTRACT - Reconstruct the original file from the compressed copy. Reads a binary file. Writes into a text file.
Write the output in the indicated *dest_file*.

`compressor.core.retrieve_table` (*dest_file*: `io`) → `dict`
Read the binary file, and return the translation table as a reversed dictionary.

`compressor.core.save_compressed_file` (*filename*: `str`, *table*: `dict`, *checksum*: `int`, *dest_file*: `str = ''`) → `None`
Given the original file by its *filename*, save a new one. *table* contains the new codes for each character on *filename*.

`compressor.core.save_table` (*dest_file*: `io`, *table*: `dict`) → `None`

Store the table in the destination file. *c*: char *L*: code of *c* (unsigned Long)

Parameters

- **dest_file** (`io`) – opened file where to write the *table*.
- **table** (`dict`) – Mapping table with the chars and their codes.

2.1.6 functions module

CHAPTER 3

Indices and tables

- `genindex`
- `modindex`
- `search`

C

- `compressor`, 5
- `compressor.cli`, 7
- `compressor.core`, 8
- `compressor.lib`, 5

A

`argument_parser()` (in module *compressor.cli*), 7

C

`compress_and_save_content()` (in module *compressor.core*), 8

`compress_file()` (in module *compressor.lib*), 5

`compressor` (module), 5

`compressor.cli` (module), 7

`compressor.core` (module), 8

`compressor.lib` (module), 5

`create_tree_code()` (in module *compressor.core*), 8

D

`decode_file_content()` (in module *compressor.core*), 8

M

`main()` (in module *compressor.cli*), 7

`main_engine()` (in module *compressor.cli*), 7

O

`open_text_file()` (in module *compressor.core*), 8

`open_text_file()` (in module *compressor.lib*), 5

P

`parse_arguments()` (in module *compressor.cli*), 7

`parse_tree_code()` (in module *compressor.core*), 9

`process_frequencies()` (in module *compressor.core*), 10

`process_line_compression()` (in module *compressor.core*), 10

R

`retrieve_compressed_file()` (in module *compressor.core*), 10

`retrieve_table()` (in module *compressor.core*), 10

S

`save_compressed_file()` (in module *compressor.core*), 10

`save_table()` (in module *compressor.core*), 10