Package 'zucsv'

November 24, 2026

Title Read CSV Files Using the 'zsv' Parser

Version 0.1.1

Description Reads comma-separated and other delimited text files into base R data frames using the 'zsv' C parser <https://github.com/liquidaty/zsv> as its native backend. Handles quoted fields, embedded delimiters and newlines, escaped quotes, and both LF and CRLF line endings, and infers logical, integer, double, and character column types. Column types can also be given explicitly, in which case a value that cannot be converted is an error rather than a silently missing value. The 'zsv' sources are bundled, so there are no runtime dependencies.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.2.0)

URL https://github.com/pedrobtz/zucsv

BugReports https://github.com/pedrobtz/zucsv/issues

Suggests testthat (>= 3.0.0)

Config/testthat/edition 3

Config/Needs/website rmarkdown, data.table, vroom

Config/roxygen2/version 8.1.1

Language en-US

NeedsCompilation yes

Author Pedro Baltazar [aut, cre, cph],
Tai Chi Minh Ralph Eastwood [ctb, cph] (Author of the bundled 'zsv' library),
Matt Wong [ctb] (Author of the bundled 'zsv' library),
Guarnerix Inc dba Liquidaty [cph] (Copyright holder of the bundled 'zsv' library),
zsv contributors [ctb, cph] (Contributors to the bundled 'zsv' library),
Max Ogden [ctb, cph] (Author of the 'csv-spectrum' test fixtures)

Maintainer Pedro Baltazar <pedrobtz@gmail.com>

Repository CRAN

Date/Publication 2026-11-24 15:00:02 UTC

Contents

read_csv 2
sniff_csv 4

Index **6**

`read_csv` *Read a CSV file*

Description

Reads a delimited text file into a base R data frame, using the bundled `zsv` C parser. The file is read twice: once to establish the shape and validate it, once to fill columns that were allocated at their exact final length.

Usage

```
read_csv(
  file = NULL,
  header = TRUE,
  delimiter = ",",
  na = c("", "NA"),
  col_types = NULL,
  text = NULL
)
```

Arguments

`file`	Path to a local file, as a single string. `~` is expanded. Supply this or `text`, not both.
`header`	If `TRUE` (the default), the first record supplies the column names. If `FALSE`, every record is data and columns are named `V1`, `V2`, ... If `NA`, detect: the first record is treated as names unless one of its unquoted fields looks like a number or a logical, in which case it is data. A quoted field is always text, and so always counts as a name: `"2024"` and `"TRUE"` are column names where bare `2024` and `TRUE` are data. Header values are always text: they are never matched against `na` and never type-inferred.
`delimiter`	The field delimiter, as a single ASCII character. Defaults to `","`. A newline, carriage return, form feed or double quote is not allowed.
`na`	Character vector of cell values to read as missing, or `NULL` to disable missing-value matching. Matching happens after unquoting and is exact: no whitespace is trimmed, so `NA` and `"NA"` are both missing under the default but `" NA"` is not.
`col_types`	`NULL` to infer each column's type, or a character vector of `"logical"`, `"integer"`, `"double"` or `"character"`. A single value applies to every column; otherwise give one per column. A cell that cannot be converted to a forced type is an error, not a missing value.

`text` The CSV itself, as a character vector, instead of a path. A vector of several elements is joined with newlines, so each element is one line — `readLines()` output reads back unchanged, except that a CR inside a quoted field comes back as LF: `readLines()` ends a line at a CR, and the join puts an LF there. The string's declared encoding is honoured and converted to UTF-8, so a latin1 string reads correctly where a latin1 *file* is still an error.

Details

Some behavior worth knowing before you rely on it:

- **Blank lines are skipped**, wherever they appear, and do not count as rows. A line holding only whitespace or only delimiters is *not* blank: it is a record of empty cells and must match the table's width. In a single-column file, write an intentional empty cell as `""` — a bare empty line is skipped.
- **Every record must have the same width.** A mismatch is an error; short rows are not padded.
- **A record may be at most 8,388,607 bytes**, counting delimiters and quotes but not its line terminator, and a file at most 65,536 columns. A record over either limit is an error naming it.
- **Column names are kept exactly as they appear**, including duplicates and empty strings. No name repair is performed.
- **Text must be valid UTF-8.** A byte sequence that is not, or an embedded NUL, is an error rather than something passed through to fail later. A UTF-8 BOM is ignored.
- **Bytes inside a quoted field are kept as they are.** A CR LF pair or a lone CR in a quoted field stays in the cell, whatever line ending the file uses; only the record terminator is removed.
- `header = NA` **looks at the first record only**, and at nothing else: not the rest of the file, not `col_types`, and not `na`. It therefore reaches a different verdict from `data.table::fread()` and DuckDB, which compare the first record against the types of the rest, on three shapes — and the row count moves in both directions:
  - an unquoted header of numeric-looking names over numeric columns, such as `id,2024,2025` above `1,5,6`, reads as data, so the name record becomes a row and you get **one row more**;
  - a first record that is entirely missing values, `NA,NA,NA` above `1,2,3`, reads as a header, costing **one row**;
  - a headerless first record of entirely quoted numbers, `"1","2"` above `"3","4"`, reads as a header, costing **one row**.

  Detection is a convenience for files whose shape you do not know. When you do know, pass `header = TRUE` or `header = FALSE` and none of this applies.

Value

A data.frame with one column per field. Names come from the first record when `header = TRUE`, or when `header = NA` and that record is detected as names; otherwise they are `V1`, `V2`, ... An empty file, or one of nothing but blank lines, gives a data frame with zero rows and zero columns; a header-only file gives zero rows but one column per header field.

See Also

`sniff_csv()`, to see the header verdict and inferred types before committing to them — and to pin them down so nothing is inferred on the next run.

Examples

```
# A small CSV given directly, rather than from a file
read_csv(text = "a,b\n1,2")

# A character vector is one line per element
read_csv(text = c("id,name", "1,Ada", "2,Linus"))

path <- tempfile(fileext = ".csv")
write.csv(
  data.frame(id = 1:3, name = c("Ada", "Linus", "Grace")),
  path,
  row.names = FALSE
)

read_csv(path)

# Force a column's type rather than inferring it
read_csv(path, col_types = c("character", "character"))

# Detect whether the first record is a header
read_csv(path, header = NA)

unlink(path)
```

`sniff_csv` *Report how a CSV file would be read*

Description

Runs the first of `read_csv()`'s two passes and returns what it learned — the header verdict, the shape, the column names and the inferred column types — without building any columns. Use it when you want to see the decisions before committing to them, or to pin them down so a later run cannot drift:

Usage

```
sniff_csv(file = NULL, delimiter = ",", na = c("", "NA"), text = NULL)
```

Arguments

`file`	Path to a local file, as a single string. `~` is expanded. Supply this or `text`, not both.
`delimiter`	The field delimiter, as a single ASCII character. Defaults to `","`. Not detected — `zucsv` never guesses the delimiter.
`na`	Character vector of cell values to read as missing, or `NULL` to disable missing-value matching. Affects the inferred types, so pass here whatever you will pass to `read_csv()`.
`text`	The CSV itself, as a character vector, instead of a path, with the same meaning as in `read_csv()`.

Details

```
s <- sniff_csv(path)
read_csv(path, header = s$header, col_types = s$col_types)
```

That second call reads the file strictly: nothing about it is inferred, so it gives the same answer next month even if the data changes shape.

Sniffing reads the whole file and applies exactly the validation `read_csv()` applies, so a ragged row, an embedded NUL or invalid UTF-8 is an error here too. If `sniff_csv()` succeeds, `read_csv()` on the same file will not fail on content.

It costs one pass rather than two, but that pass is the expensive one: do not call it before every read as a matter of course. Its purpose is to let you *stop* inferring, not to infer twice.

Value

A list with five elements:

`header` `TRUE` if the first record is names, `FALSE` if it is data, by the rule documented under `header` in `read_csv()`.

`ncol` Number of columns.

`nrow` Number of data rows — exact, not an estimate, because the pass counts every record.

`col_names` The names the file would produce: the first record when `header` is `TRUE`, otherwise `V1`, `V2`, ...

`col_types` Inferred type of each column, suitable for passing straight back as `col_types`.

An empty file gives `header = FALSE`, zero columns and zero rows.

See Also

`read_csv()`

Examples

```
path <- tempfile(fileext = ".csv")
writeLines(c("id,name,score", "1,Ada,9.5", "2,Grace,8"), path)

sniff_csv(path)

# Pin the decisions, then read without inferring anything
s <- sniff_csv(path)
read_csv(path, header = s$header, col_types = s$col_types)

unlink(path)
```

Index

data.frame, *3*

read_csv, 2
read_csv(), *4*, *5*

sniff_csv, 4
sniff_csv(), *3*