# Package 'readr'

August 9, 2021

**Title** Read Rectangular Text Data

**Version** 2.0.1

**Description** The goal of 'readr' is to provide a fast and
friendly way to read rectangular data (like 'csv', 'tsv', and 'fwf').
It is designed to flexibly parse many types of data found in the wild,
while still cleanly failing when data unexpectedly changes.

**License** MIT + file LICENSE

**URL** https://readr.tidyverse.org,
https://github.com/tidyverse/readr

**BugReports** https://github.com/tidyverse/readr/issues

**Depends** R (>= 3.1)

**Imports** cli,
clipr,
crayon,
hms (>= 0.4.1),
methods,
rlang,
R6,
tibble,
vroom (>= 1.5.4),
utils,
lifecycle (>= 0.2.0)

**Suggests** covr,
curl,
dplyr,
knitr,
rmarkdown,
spelling,
stringi,
testthat,
tzdb (>= 0.1.1),
waldo,
withr,
xml2

**LinkingTo** cpp11,
tzdb (>= 0.1.1)

**VignetteBuilder** knitr

**Config/Needs/website** pkgdown, tidyverse, tidyverse/tidytemplate

**Config/testthat/edition** 3

**Config/testthat/parallel** false

**Encoding** UTF-8

**Language** en-US

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.1.1

**SystemRequirements** C++11

**RdMacros** lifecycle

# R topics documented:

---

`clipboard` *Returns values from the clipboard*

---

## Description

This is useful in the `read_delim()` functions to read from the clipboard.

## Usage

```
clipboard()
```

## See Also

read_delim

---

`cols` *Create column specification*

---

## Description

`cols()` includes all columns in the input data, guessing the column types as the default. `cols_only()` includes only the columns you explicitly specify, skipping the rest. In general you can substitute `list()` for `cols()` without changing the behavior.

## Usage

```
cols(..., .default = col_guess())

cols_only(...)
```

## Arguments

| | |
|---|---|
| `...` | Either column objects created by col_*(), or their abbreviated character names (as described in the `col_types` argument of `read_delim()`). If you're only overriding a few columns, it's best to refer to columns by name. If not named, the column types must match the column names exactly. |
| `.default` | Any named columns not explicitly overridden in `...` will be read with this column type. |

## Details

The available specifications are: (with string abbreviations in brackets)

- `col_logical()` [l], containing only `T`, `F`, `TRUE` or `FALSE`.
- `col_integer()` [i], integers.
- `col_double()` [d], doubles.
- `col_character()` [c], everything else.
- `col_factor(levels,ordered)` [f], a fixed set of values.
- `col_date(format = "")` [D]: with the locale's `date_format`.
- `col_time(format = "")` [t]: with the locale's `time_format`.
- `col_datetime(format = "")` [T]: ISO8601 date times
- `col_number()` [n], numbers containing the `grouping_mark`
- `col_skip()` [_, -], don't import this column.
- `col_guess()` [?], parse using the "best" type based on the input.

## See Also

Other parsers: `col_skip()`, `cols_condense()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

## Examples

```
cols(a = col_integer())
cols_only(a = col_integer())

# You can also use the standard abbreviations
cols(a = "i")
cols(a = "i", b = "d", c = "_")

# You can also use multiple sets of column definitions by combining
# them like so:

t1 <- cols(
  column_one = col_integer(),
  column_two = col_number()
)

t2 <- cols(
  column_three = col_character()
)

t3 <- t1
t3$cols <- c(t1$cols, t2$cols)
t3
```

## cols_condense *Examine the column specifications for a data frame*

### Description

`cols_condense()` takes a spec object and condenses its definition by setting the default column type to the most frequent type and only listing columns with a different type.

`spec()` extracts the full column specification from a tibble created by readr.

### Usage

```
cols_condense(x)

spec(x)
```

### Arguments

| | |
|---|---|
| `x` | The data frame object to extract from |

### Value

A col_spec object.

### See Also

Other parsers: `col_skip()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

### Examples

```
df <- read_csv(readr_example("mtcars.csv"))
s <- spec(df)
s

cols_condense(s)
```

## col_skip *Skip a column*

### Description

Use this function to ignore a column when reading in a file. To skip all columns not otherwise specified, use `cols_only()`.

### Usage

```
col_skip()
```

### See Also

Other parsers: `cols_condense()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

## count_fields *Count the number of fields in each line of a file*

### Description

This is useful for diagnosing problems with functions that fail to parse correctly.

### Usage

```
count_fields(file, tokenizer, skip = 0, n_max = -1L)
```

### Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| tokenizer | A tokenizer that specifies how to break the file up into fields, e.g., tokenizer_csv(), tokenizer_fwf() |
| skip | Number of lines to skip before reading data. |
| n_max | Optionally, maximum number of rows to count fields for. |

### Examples

```
count_fields(readr_example("mtcars.csv"), tokenizer_csv())
```

## date_names *Create or retrieve date names*

### Description

When parsing dates, you often need to know how weekdays of the week and months are represented as text. This pair of functions allows you to either create your own, or retrieve from a standard list. The standard list is derived from ICU (http://site.icu-project.org) via the stringi package.

### Usage

```
date_names(mon, mon_ab = mon, day, day_ab = day, am_pm = c("AM", "PM"))

date_names_lang(language)

date_names_langs()
```

### Arguments

| | |
|---|---|
| `mon, mon_ab` | Full and abbreviated month names. |
| `day, day_ab` | Full and abbreviated week day names. Starts with Sunday. |
| `am_pm` | Names used for AM and PM. |
| `language` | A BCP 47 locale, made up of a language and a region, e.g. `"en_US"` for American English. See `date_names_langs()` for a complete list of available locales. |

### Examples

```
date_names_lang("en")
date_names_lang("ko")
date_names_lang("fr")
```

---

## `edition_get` *Retrieve the currently active edition*

---

### Description

Retrieve the currently active edition

### Usage

```
edition_get()
```

### Value

An integer corresponding to the currently active edition.

---

## `format_delim` *Convert a data frame to a delimited string*

---

### Description

These functions are equivalent to `write_csv()` etc., but instead of writing to disk, they return a string.

### Usage

```
format_delim(
  x,
  delim,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  quote_escape = deprecated()
)
```

```
format_csv(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  quote_escape = deprecated()
)

format_csv2(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  quote_escape = deprecated()
)

format_tsv(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  quote_escape = deprecated()
)
```

## Arguments

| | |
|---|---|
| x | A data frame. |
| delim | Delimiter used to separate values. Defaults to " " for write_delim(), "," for write_excel_csv() and ";" for write_excel_csv2(). Must be a single character. |
| na | String used for missing values. Defaults to NA. Missing values will never be quoted; strings with the same value as na will always be quoted. |
| append | If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created. |
| col_names | If FALSE, column names will not be included at the top of the file. If TRUE, column names will be included. If not specified, col_names will take the opposite value given to append. |
| quote | How to handle fields which contain characters that need to be quoted.<br>• needed - Only quote fields which need them.<br>• all - Quote all fields.<br>• none - Never quote fields. |

escape — The type of escape to use when quotes are in the data.

- `double` - quotes are escaped by doubling them.
- `backslash` - quotes are escaped by a preceding backslash.
- `none` - quotes are not escaped.

eol — The end of line character to use. Most commonly either `"\n"` for Unix style newlines, or `"\r\n"` for Windows style newlines.

quote_escape — **[Deprecated]**, use the `escape` argument instead.

## Value

A string.

## Output

Factors are coerced to character. Doubles are formatted to a decimal string using the grisu3 algorithm. `POSIXct` values are formatted as ISO8601 with a UTC timezone *Note:* `POSIXct` *objects in local or non-UTC timezones will be converted to UTC time before writing.*

All columns are encoded as UTF-8. `write_excel_csv()` and `write_excel_csv2()` also include a UTF-8 Byte order mark which indicates to Excel the csv is UTF-8 encoded.

`write_excel_csv2()` and `write_csv2` were created to allow users with different locale settings to save .csv files using their default settings (e.g. ; as the column separator and , as the decimal separator). This is common in some European countries.

Values are only quoted if they contain a comma, quote or newline.

The write_*() functions will automatically compress outputs if an appropriate extension is given. Three extensions are currently supported: `.gz` for gzip compression, `.bz2` for bzip2 compression and `.xz` for lzma compression. See the examples for more information.

## References

Florian Loitsch, Printing Floating-Point Numbers Quickly and Accurately with Integers, PLDI '10, http://www.cs.tufts.edu/~nr/cs257/archive/florian-loitsch/printf.pdf

## Examples

```
# format_()* functions are useful for testing and reprexes
cat(format_csv(mtcars))
cat(format_tsv(mtcars))
cat(format_delim(mtcars, ";"))

# Specifying missing values
df <- data.frame(x = c(1, NA, 3))
format_csv(df, na = "missing")

# Quotes are automatically added as needed
df <- data.frame(x = c("a ", '"', ",", "\n"))
cat(format_csv(df))
```

## guess_encoding *Guess encoding of file*

### Description

Uses `stringi::stri_enc_detect()`: see the documentation there for caveats.

### Usage

```
guess_encoding(file, n_max = 10000, threshold = 0.2)
```

### Arguments

| | |
|---|---|
| `file` | A character string specifying an input as specified in `datasource()`, a raw vector, or a list of raw vectors. |
| `n_max` | Number of lines to read. If `n_max` is -1, all lines in file will be read. |
| `threshold` | Only report guesses above this threshold of certainty. |

### Value

A tibble

### Examples

```
guess_encoding(readr_example("mtcars.csv"))
guess_encoding(read_lines_raw(readr_example("mtcars.csv")))
guess_encoding(read_file_raw(readr_example("mtcars.csv")))

guess_encoding("a\n\u00b5\u00b5")
```

## locale *Create locales*

### Description

A locale object tries to capture all the defaults that can vary between countries. You set the locale in once, and the details are automatically passed on down to the columns parsers. The defaults have been chosen to match R (i.e. US English) as closely as possible. See `vignette("locales")` for more details.

### Usage

```
locale(
  date_names = "en",
  date_format = "%AD",
  time_format = "%AT",
  decimal_mark = ".",
  grouping_mark = ",",
  tz = "UTC",
  encoding = "UTF-8",
```

```
  asciify = FALSE
)

default_locale()
```

## Arguments

| | |
|---|---|
| `date_names` | Character representations of day and month names. Either the language code as string (passed on to `date_names_lang()`) or an object created by `date_names()`. |
| `date_format, time_format` | Default date and time formats. |
| `decimal_mark, grouping_mark` | Symbols used to indicate the decimal place, and to chunk larger numbers. Decimal mark can only be , or . . |
| `tz` | Default tz. This is used both for input (if the time zone isn't present in individual strings), and for output (to control the default display). The default is to use "UTC", a time zone that does not use daylight savings time (DST) and hence is typically most useful for data. The absence of time zones makes it approximately 50x faster to generate UTC times than any other time zone.<br>Use `""` to use the system default time zone, but beware that this will not be reproducible across systems.<br>For a complete list of possible time zones, see `OlsonNames()`. Americans, note that "EST" is a Canadian time zone that does not have DST. It is *not* Eastern Standard Time. It's better to use "US/Eastern", "US/Central" etc. |
| `encoding` | Default encoding. This only affects how the file is read - readr always converts the output to UTF-8. |
| `asciify` | Should diacritics be stripped from date names and converted to ASCII? This is useful if you're dealing with ASCII data where the correct spellings have been lost. Requires the **stringi** package. |

## Examples

```
locale()
locale("fr")

# South American locale
locale("es", decimal_mark = ",")
```

---

`melt_delim` *Return melted data for each token in a delimited file (including csv & tsv)*

---

## Description

**[Superseded]** This function has been superseded in readr and moved to the meltr package.

## Usage

```
melt_delim(
  file,
  delim,
  quote = "\"",
  escape_backslash = FALSE,
  escape_double = TRUE,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)

melt_csv(
  file,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)

melt_csv2(
  file,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)

melt_tsv(
  file,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
```

```
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)
```

## Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| delim | Single character used to separate fields within a record. |
| quote | Single character used to quote strings. |
| escape_backslash | Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \\n. |
| escape_double | Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, \". |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| quoted_na | Should missing values inside quotes be treated as missing values (the default) or strings. |
| comment | A string used to identify comments. Any text after the comment characters will be silently ignored. |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |
| skip | Number of lines to skip before reading data. If comment is supplied any commented lines are ignored *after* skipping. |
| n_max | Maximum number of lines to read. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE. |
| skip_empty_rows | Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns. |

## Details

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

`melt_csv()` and `melt_tsv()` are special cases of the general `melt_delim()`. They're useful for reading the most common types of flat file data, comma separated values and tab separated values, respectively. `melt_csv2()` uses ; for the field separator and , for the decimal point. This is common in some European countries.

## Value

A `tibble()` of four columns:

- `row`, the row that the token comes from in the original file
- `col`, the column that the token comes from in the original file
- `data_type`, the data type of the token, e.g. `"integer"`, `"character"`, `"date"`, guessed in a similar way to the `guess_parser()` function.
- `value`, the token itself as a character string, unchanged from its representation in the original file.

If there are parsing problems, a warning tells you how many, and you can retrieve the details with `problems()`.

## See Also

`read_delim()` for the conventional way to read rectangular data from delimited files.

## Examples

```
# Input sources -------------------------------------------------------------
# Read from a path
melt_csv(readr_example("mtcars.csv"))
melt_csv(readr_example("mtcars.csv.zip"))
melt_csv(readr_example("mtcars.csv.bz2"))
## Not run:
melt_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")

## End(Not run)

# Or directly from a string (must contain a newline)
melt_csv("x,y\n1,2\n3,4")

# To import empty cells as 'empty' rather than `NA`
melt_csv("x,y\n,NA,\"\",''", na = "NA")

# File types ----------------------------------------------------------------
melt_csv("a,b\n1.0,2.0")
melt_csv2("a;b\n1,0;2,0")
melt_tsv("a\tb\n1.0\t2.0")
melt_delim("a|b\n1.0|2.0", delim = "|")
```

`melt_fwf` *Return melted data for each token in a fixed width file*

## Description

**[Superseded]** This function has been superseded in readr and moved to the meltr package.

## Usage

```
melt_fwf(
  file,
  col_positions,
  locale = default_locale(),
  na = c("", "NA"),
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)
```

## Arguments

| | |
|---|---|
| `file` | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of `clipboard()` will read from the system clipboard. |
| `col_positions` | Column positions, as created by `fwf_empty()`, `fwf_widths()` or `fwf_positions()`. To read in only selected fields, use `fwf_positions()`. If the width of the last column is variable (a ragged fwf file), supply the last end position as NA. |
| `locale` | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| `na` | Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values. |
| `comment` | A string used to identify comments. Any text after the comment characters will be silently ignored. |
| `trim_ws` | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |
| `skip` | Number of lines to skip before reading data. |

| | |
|---|---|
| n_max | Maximum number of lines to read. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE. |
| skip_empty_rows | Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns. |

## Details

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

melt_fwf() parses each token of a fixed width file into a single row, but it still requires that each field is in the same in every row of the source file.

## See Also

melt_table() to melt fixed width files where each column is separated by whitespace, and read_fwf() for the conventional way to read rectangular data from fixed width files.

## Examples

```
fwf_sample <- readr_example("fwf-sample.txt")
cat(read_lines(fwf_sample))

# You can specify column positions in several ways:
# 1. Guess based on position of empty columns
melt_fwf(fwf_sample, fwf_empty(fwf_sample, col_names = c("first", "last", "state", "ssn")))
# 2. A vector of field widths
melt_fwf(fwf_sample, fwf_widths(c(20, 10, 12), c("name", "state", "ssn")))
# 3. Paired vectors of start and end positions
melt_fwf(fwf_sample, fwf_positions(c(1, 30), c(10, 42), c("name", "ssn")))
# 4. Named arguments with start and end positions
melt_fwf(fwf_sample, fwf_cols(name = c(1, 10), ssn = c(30, 42)))
# 5. Named arguments with column widths
melt_fwf(fwf_sample, fwf_cols(name = 20, state = 10, ssn = 12))
```

---

# melt_table *Return melted data for each token in a whitespace-separated file*

---

## Description

**[Superseded]** This function has been superseded in readr and moved to the meltr package.

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

melt_table() and melt_table2() are designed to read the type of textual data where each column is separated by one (or more) columns of space.

melt_table2() allows any number of whitespace characters between columns, and the lines can be of different lengths.

melt_table() is more strict, each line must be the same length, and each field is in the same position in every line. It first finds empty columns and then parses like a fixed width file.

## Usage

```
melt_table(
  file,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  comment = "",
  skip_empty_rows = FALSE
)

melt_table2(
  file,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  comment = "",
  skip_empty_rows = FALSE
)
```

## Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| skip | Number of lines to skip before reading data. |
| n_max | Maximum number of lines to read. |
| guess_max | Maximum number of lines to use for guessing column types. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE. |
| comment | A string used to identify comments. Any text after the comment characters will be silently ignored. |

skip_empty_rows
: Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.

## See Also

melt_fwf() to melt fixed width files where each column is not separated by whitespace. melt_fwf() is also useful for reading tabular data with non-standard formatting. read_table() is the conventional way to read tabular data from whitespace-separated files.

## Examples

```
# One corner from http://www.masseyratings.com/cf/compare.htm
massey <- readr_example("massey-rating.txt")
cat(read_file(massey))
melt_table(massey)

# Sample of 1978 fuel economy data from
# http://www.fueleconomy.gov/feg/epadata/78data.zip
epa <- readr_example("epa78.txt")
cat(read_file(epa))
melt_table(epa)
```

---

# parse_atomic *Parse logicals, integers, and reals*

---

## Description

Use parse_*() if you have a character vector you want to parse. Use col_*() in conjunction with a read_*() function to parse the values as they're read in.

## Usage

```
parse_logical(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)

parse_integer(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)

parse_double(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)

parse_character(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)

col_logical()

col_integer()

col_double()

col_character()
```

### Arguments

| | |
|---|---|
| x | Character vector of values to parse. |
| na | Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |

### See Also

Other parsers: `col_skip()`, `cols_condense()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_number()`, `parse_vector()`

### Examples

```
parse_integer(c("1", "2", "3"))
parse_double(c("1", "2", "3.123"))
parse_number("$1,123,456.00")

# Use locale to override default decimal and grouping marks
es_MX <- locale("es", decimal_mark = ",")
parse_number("$1.123.456,00", locale = es_MX)

# Invalid values are replaced with missing values with a warning.
x <- c("1", "2", "3", "-")
parse_double(x)
# Or flag values as missing
parse_double(x, na = "-")
```

---

## parse_datetime *Parse date/times*

---

### Description

Parse date/times

### Usage

```
parse_datetime(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)

parse_date(
  x,
```

```
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)

parse_time(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)

col_datetime(format = "")

col_date(format = "")

col_time(format = "")
```

## Arguments

| | |
|---|---|
| x | A character vector of dates to parse. |
| format | A format specification, as described below. If set to "", date times are parsed as ISO8601, dates and times used the date and time formats specified in the locale().<br>Unlike strptime(), the format specification must match the complete string. |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |

## Value

A POSIXct() vector with tzone attribute set to tz. Elements that could not be parsed (or did not generate valid dates) will be set to NA, and a warning message will inform you of the total number of failures.

## Format specification

readr uses a format specification similar to strptime(). There are three types of element:

1. Date components are specified with "%" followed by a letter. For example "%Y" matches a 4 digit year, "%m", matches a 2 digit month and "%d" matches a 2 digit day. Month and day default to 1, (i.e. Jan 1st) if not present, for example if only a year is given.
2. Whitespace is any sequence of zero or more whitespace characters.
3. Any other character is matched exactly.

`parse_datetime()` recognises the following format specifications:

- Year: "%Y" (4 digits). "%y" (2 digits); 00-69 -> 2000-2069, 70-99 -> 1970-1999.
- Month: "%m" (2 digits), "%b" (abbreviated name in current locale), "%B" (full name in current locale).
- Day: "%d" (2 digits), "%e" (optional leading space), "%a" (abbreviated name in current locale).
- Hour: "%H" or "%I" or "%h", use I (and not H) with AM/PM, use h (and not H) if your times represent durations longer than one day.
- Minutes: "%M"
- Seconds: "%S" (integer seconds), "%OS" (partial seconds)
- Time zone: "%Z" (as name, e.g. "America/Chicago"), "%z" (as offset from UTC, e.g. "+0800")
- AM/PM indicator: "%p".
- Non-digits: "%." skips one non-digit character, "%+" skips one or more non-digit characters, "%*" skips any number of non-digits characters.
- Automatic parsers: "%AD" parses with a flexible YMD parser, "%AT" parses with a flexible HMS parser.
- Time since the Unix epoch: "%s" decimal seconds since the Unix epoch.
- Shortcuts: "%D" = "%m/%d/%y", "%F" = "%Y-%m-%d", "%R" = "%H:%M", "%T" = "%H:%M:%S", "%x" = "%y/%m/%d".

### ISO8601 support

Currently, readr does not support all of ISO8601. Missing features:

- Week & weekday specifications, e.g. "2013-W05", "2013-W05-10".
- Ordinal dates, e.g. "2013-095".
- Using commas instead of a period for decimal separator.

The parser is also a little laxer than ISO8601:

- Dates and times can be separated with a space, not just T.
- Mostly correct specifications like "2009-05-19 14:" and "200912-01" work.

### See Also

Other parsers: `col_skip()`, `cols_condense()`, `cols()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

### Examples

```
# Format strings ---------------------------------------------------------
parse_datetime("01/02/2010", "%d/%m/%Y")
parse_datetime("01/02/2010", "%m/%d/%Y")
# Handle any separator
parse_datetime("01/02/2010", "%m%.%d%.%Y")

# Dates look the same, but internally they use the number of days since
# 1970-01-01 instead of the number of seconds. This avoids a whole lot
# of troubles related to time zones, so use if you can.
parse_date("01/02/2010", "%d/%m/%Y")
```

```
parse_date("01/02/2010", "%m/%d/%Y")

# You can parse timezones from strings (as listed in OlsonNames())
parse_datetime("2010/01/01 12:00 US/Central", "%Y/%m/%d %H:%M %Z")
# Or from offsets
parse_datetime("2010/01/01 12:00 -0600", "%Y/%m/%d %H:%M %z")

# Use the locale parameter to control the default time zone
# (but note UTC is considerably faster than other options)
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M",
  locale = locale(tz = "US/Central")
)
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M",
  locale = locale(tz = "US/Eastern")
)

# Unlike strptime, the format specification must match the complete
# string (ignoring leading and trailing whitespace). This avoids common
# errors:
strptime("01/02/2010", "%d/%m/%y")
parse_datetime("01/02/2010", "%d/%m/%y")

# Failures --------------------------------------------------------
parse_datetime("01/01/2010", "%d/%m/%Y")
parse_datetime(c("01/ab/2010", "32/01/2010"), "%d/%m/%Y")

# Locales ---------------------------------------------------------
# By default, readr expects English date/times, but that's easy to change'
parse_datetime("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
parse_datetime("1 enero 2015", "%d %B %Y", locale = locale("es"))

# ISO8601 ---------------------------------------------------------
# With separators
parse_datetime("1979-10-14")
parse_datetime("1979-10-14T10")
parse_datetime("1979-10-14T10:11")
parse_datetime("1979-10-14T10:11:12")
parse_datetime("1979-10-14T10:11:12.12345")

# Without separators
parse_datetime("19791014")
parse_datetime("19791014T101112")

# Time zones
us_central <- locale(tz = "US/Central")
parse_datetime("1979-10-14T1010", locale = us_central)
parse_datetime("1979-10-14T1010-0500", locale = us_central)
parse_datetime("1979-10-14T1010Z", locale = us_central)
# Your current time zone
parse_datetime("1979-10-14T1010", locale = locale(tz = ""))
```

---

## parse_factor *Parse factors*

---

## Description

parse_factor is similar to factor(), but will generate warnings if elements of x are not found in levels.

## Usage

```
parse_factor(
  x,
  levels = NULL,
  ordered = FALSE,
  na = c("", "NA"),
  locale = default_locale(),
  include_na = TRUE,
  trim_ws = TRUE
)

col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)
```

## Arguments

| | |
|---|---|
| x | Character vector of values to parse. |
| levels | Character vector providing set of allowed levels. if NULL, will generate levels based on the unique values of x, ordered by order of appearance in x. |
| ordered | Is it an ordered factor? |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| include_na | If NA are present, include as an explicit factor to level? |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |

## See Also

Other parsers: col_skip(), cols_condense(), cols(), parse_datetime(), parse_guess(), parse_logical(), parse_number(), parse_vector()

## Examples

```
parse_factor(c("a", "b"), letters)

x <- c("cat", "dog", "caw")
levels <- c("cat", "dog", "cow")

# Base R factor() silently converts unknown levels to NA
x1 <- factor(x, levels)

# parse_factor generates a warning & problems
x2 <- parse_factor(x, levels)
```

```
# Using an argument of `NULL` will generate levels based on values of `x`
x2 <- parse_factor(x, levels = NULL)
```

---

`parse_guess` *Parse using the "best" type*

---

## Description

`parse_guess()` returns the parser vector; `guess_parser()` returns the name of the parser. These functions use a number of heuristics to determine which type of vector is "best". Generally they try to err of the side of safety, as it's straightforward to override the parsing choice if needed.

## Usage

```
parse_guess(
  x,
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE,
  guess_integer = FALSE
)

col_guess()

guess_parser(
  x,
  locale = default_locale(),
  guess_integer = FALSE,
  na = c("", "NA")
)
```

## Arguments

| | |
|---|---|
| `x` | Character vector of values to parse. |
| `na` | Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values. |
| `locale` | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| `trim_ws` | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |
| `guess_integer` | If `TRUE`, guess integer types for whole numbers, if `FALSE` guess numeric type for all numbers. |

## See Also

Other parsers: `col_skip()`, `cols_condense()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_logical()`, `parse_number()`, `parse_vector()`

## Examples

```
# Logical vectors
parse_guess(c("FALSE", "TRUE", "F", "T"))

# Integers and doubles
parse_guess(c("1", "2", "3"))
parse_guess(c("1.6", "2.6", "3.4"))

# Numbers containing grouping mark
guess_parser("1,234,566")
parse_guess("1,234,566")

# ISO 8601 date times
guess_parser(c("2010-10-10"))
parse_guess(c("2010-10-10"))
```

---

`parse_number` *Parse numbers, flexibly*

---

## Description

This drops any non-numeric characters before or after the first number. The grouping mark specified by the locale is ignored inside the number.

## Usage

```
parse_number(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)

col_number()
```

## Arguments

| | |
|---|---|
| `x` | Character vector of values to parse. |
| `na` | Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values. |
| `locale` | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| `trim_ws` | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |

## Value

A numeric vector (double) of parsed numbers.

## See Also

Other parsers: `col_skip()`, `cols_condense()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_vector()`

## Examples

```
## These all return 1000
parse_number("$1,000") ## leading $ and grouping character , ignored
parse_number("euro1,000") ## leading non-numeric euro ignored

parse_number("1,234.56")
## explicit locale specifying European grouping and decimal marks
parse_number("1.234,56", locale = locale(decimal_mark = ",", grouping_mark = "."))
## SI/ISO 31-0 standard spaces for number grouping
parse_number("1 234.56", locale = locale(decimal_mark = ".", grouping_mark = " "))

## Specifying strings for NAs
parse_number(c("1", "2", "3", "NA"))
parse_number(c("1", "2", "3", "NA", "Nothing"), na = c("NA", "Nothing"))
```

---

`problems` *Retrieve parsing problems*

---

## Description

Readr functions will only throw an error if parsing fails in an unrecoverable way. However, there are lots of potential problems that you might want to know about - these are stored in the `problems` attribute of the output, which you can easily access with this function. `stop_for_problems()` will throw an error if there are any parsing problems: this is useful for automated scripts where you want to throw an error as soon as you encounter a problem.

## Usage

```
problems(x = .Last.value)

stop_for_problems(x)
```

## Arguments

| | |
|---|---|
| `x` | An data frame (from read_*()) or a vector (from parse_*()). |

## Value

A data frame with one row for each problem and four columns:

| | |
|---|---|
| `row,col` | Row and column of problem |
| `expected` | What readr expected to find |
| `actual` | What it actually got |

## Examples

```
x <- parse_integer(c("1X", "blah", "3"))
problems(x)

y <- parse_integer(c("1", "2", "3"))
problems(y)
```

## readr_example *Get path to readr example*

### Description

readr comes bundled with a number of sample files in its `inst/extdata` directory. This function make them easy to access

### Usage

```
readr_example(file = NULL)
```

### Arguments

| | |
|---|---|
| `file` | Name of file. If `NULL`, the example files will be listed. |

### Examples

```
readr_example()
readr_example("challenge.csv")
```

## readr_threads *Determine how many threads readr should use when processing*

### Description

The number of threads returned can be set by

- The global option `readr.num_threads`
- The environment variable `VROOM_THREADS`
- The value of `parallel::detectCores()`

### Usage

```
readr_threads()
```

## read_builtin *Read built-in object from package*

### Description

Consistent wrapper around `data()` that forces the promise. This is also a stronger parallel to loading data from a file.

### Usage

```
read_builtin(x, package = NULL)
```

### Arguments

| | |
|---|---|
| x | Name (character string) of data set to read. |
| package | Name of package from which to find data set. By default, all attached packages are searched and then the 'data' subdirectory (if present) of the current working directory. |

### Value

An object of the built-in class of `x`.

### Examples

```
if (requireNamespace("dplyr")) {
  read_builtin("starwars", "dplyr")

  read_builtin("storms", "dplyr")
}
```

## read_delim *Read a delimited file (including CSV and TSV) into a tibble*

### Description

`read_csv()` and `read_tsv()` are special cases of the more general `read_delim()`. They're useful for reading the most common types of flat file data, comma separated values and tab separated values, respectively. `read_csv2()` uses ; for the field separator and , for the decimal point. This format is common in some European countries.

### Usage

```
read_delim(
  file,
  delim = NULL,
  quote = "\"",
  escape_backslash = FALSE,
  escape_double = TRUE,
  col_names = TRUE,
```

```
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

read_csv(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

read_csv2(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
```

```
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  progress = show_progress(),
  name_repair = "unique",
  num_threads = readr_threads(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

read_tsv(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  progress = show_progress(),
  name_repair = "unique",
  num_threads = readr_threads(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)
```

## Arguments

| | |
|---|---|
| `file` | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of `clipboard()` will read from the system clipboard. |
| `delim` | Single character used to separate fields within a record. |
| `quote` | Single character used to quote strings. |
| `escape_backslash` | Does the file use backslashes to escape special characters? This is more gen- |

eral than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \\n.

escape_double
: Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, \".

col_names
: Either TRUE, FALSE or a character vector of column names.

  If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.

  If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.

  Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique, see name_repair to control how this is done.

col_types
: One of NULL, a cols() specification, or a string. See vignette("readr") for more details.

  If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to increase the guess_max or supply the correct types yourself.

  Column specifications created by list() or cols() must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().

  Alternatively, you can use a compact string representation where each character represents one column:

  - c = character
  - i = integer
  - n = number
  - d = double
  - l = logical
  - f = factor
  - D = date
  - T = date time
  - t = time
  - ? = guess
  - _ or - = skip

    By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, set show_col_types = FALSE or set 'options(readr.show_col_types = FALSE).

col_select
: <tidy-select> Columns to include in the results, either by name or by numeric index. Use c() or list() to select with more than one expression and ?tidyselect::language for full details on the selection language.

id
: The name of a column in which to store the file path. This is useful when reading multiple input files and there is data in the file paths, such as the data collection date. If NULL (the default) no extra column is created.

locale
: The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

| | |
|---|---|
| na | Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values. |
| quoted_na | Should missing values inside quotes be treated as missing values (the default) or strings. |
| comment | A string used to identify comments. Any text after the comment characters will be silently ignored. |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |
| skip | Number of lines to skip before reading data. If `comment` is supplied any commented lines are ignored *after* skipping. |
| n_max | Maximum number of lines to read. |
| guess_max | Maximum number of lines to use for guessing column types. |
| name_repair | Treatment of problematic column names:<br>• `"minimal"`: No name repair or checks, beyond basic existence of names<br>• `"unique"`: Make sure names are unique and not empty<br>• `"check_unique"`: (default value), no name repair, but check they are `unique`<br>• `"universal"`: Make the names `unique` and syntactic<br>• a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R)<br>• A purrr-style anonymous function, see `rlang::as_function()`<br>This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them. |
| num_threads | The number of processing threads to use for initial parsing and lazy reading of data. If your data contains newlines within fields the parser should automatically detect this and fall back to using one thread only. However if you know your file has newlines within quoted fields it is safest to set `num_threads = 1` explicitly. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`. |
| show_col_types | If `FALSE`, do not show the guessed column types. If `TRUE` always show the column types, even if they are supplied. If `NULL` (the default) only show the column types if they are not explicitly supplied by the `col_types` argument. |
| skip_empty_rows | Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns. |
| lazy | Read values lazily? By default the file is initially only indexed and the values are read lazily when accessed. Lazy reading is useful interactively, particularly if you are only interested in a subset of the full dataset. Note, lazy reading on windows will lock the file until all the data has been read from it, if you run into this issue set `lazy = FALSE`. |

## Value

A `tibble()`. If there are parsing problems, a warning will alert you. You can retrieve the full details by calling `problems()` on your dataset.

## Examples

```
# Input sources -------------------------------------------------------------
# Read from a path
read_csv(readr_example("mtcars.csv"))
read_csv(readr_example("mtcars.csv.zip"))
read_csv(readr_example("mtcars.csv.bz2"))
## Not run:
# Including remote paths
read_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")

## End(Not run)

# Or directly from a string with `I()`
read_csv(I("x,y\n1,2\n3,4"))

# Column types --------------------------------------------------------------
# By default, readr guesses the columns types, looking at the first 1000 rows.
# You can override with a compact specification:
read_csv(I("x,y\n1,2\n3,4"), col_types = "dc")

# Or with a list of column types:
read_csv(I("x,y\n1,2\n3,4"), col_types = list(col_double(), col_character()))

# If there are parsing problems, you get a warning, and can extract
# more details with problems()
y <- read_csv(I("x\n1\n2\nb"), col_types = list(col_double()))
y
problems(y)

# File types ----------------------------------------------------------------
read_csv(I("a,b\n1.0,2.0"))
read_csv2(I("a;b\n1,0;2,0"))
read_tsv(I("a\tb\n1.0\t2.0"))
read_delim(I("a|b\n1.0|2.0"), delim = "|")
```

---

`read_file` *Read/write a complete file*

---

## Description

`read_file()` reads a complete file into a single object: either a character vector of length one, or a raw vector. `write_file()` takes a single string, or a raw vector, and writes it exactly as is. Raw vectors are useful when dealing with binary data, or if you have text data with unknown encoding.

## Usage

```
read_file(file, locale = default_locale())

read_file_raw(file)

write_file(x, file, append = FALSE, path = deprecated())
```

### Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| x | A single string, or a raw vector to write to disk. |
| append | If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created. |
| path | **[Deprecated]**, use the file argument instead. |

### Value

read_file: A length 1 character vector. read_lines_raw: A raw vector.

### Examples

```
read_file(file.path(R.home("doc"), "AUTHORS"))
read_file_raw(file.path(R.home("doc"), "AUTHORS"))

tmp <- tempfile()

x <- format_csv(mtcars[1:6, ])
write_file(x, tmp)
identical(x, read_file(tmp))

read_lines(I(x))
```

---

## read_fwf *Read a fixed width file into a tibble*

---

### Description

A fixed width file can be a very compact representation of numeric data. It's also very fast to parse, because every field is in the same place in every line. Unfortunately, it's painful to parse because you need to describe the length of every field. Readr aims to make it as easy as possible by providing a number of different ways to describe the field structure.

- fwf_empty() - Guesses based on the positions of empty columns.
- fwf_widths() - Supply the widths of the columns.
- fwf_positions() - Supply paired vectors of start and end positions.
- fwf_cols() - Supply named arguments of paired start and end positions or column widths.

## Usage

```
read_fwf(
  file,
  col_positions = fwf_empty(file, skip, n = guess_max),
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  name_repair = "unique",
  num_threads = readr_threads(),
  show_col_types = should_show_types(),
  lazy = TRUE,
  skip_empty_rows = TRUE
)

fwf_empty(
  file,
  skip = 0,
  skip_empty_rows = FALSE,
  col_names = NULL,
  comment = "",
  n = 100L
)

fwf_widths(widths, col_names = NULL)

fwf_positions(start, end = NULL, col_names = NULL)

fwf_cols(...)
```

## Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| col_positions | Column positions, as created by fwf_empty(), fwf_widths() or fwf_positions(). To read in only selected fields, use fwf_positions(). If the width of the last column is variable (a ragged fwf file), supply the last end position as NA. |

col_types
: One of NULL, a cols() specification, or a string. See vignette("readr") for more details.

  If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to increase the guess_max or supply the correct types yourself.

  Column specifications created by list() or cols() must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().

  Alternatively, you can use a compact string representation where each character represents one column:

  - c = character
  - i = integer
  - n = number
  - d = double
  - l = logical
  - f = factor
  - D = date
  - T = date time
  - t = time
  - ? = guess
  - _ or - = skip

    By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, set show_col_types = FALSE or set 'options(readr.show_col_types = FALSE).

col_select
: <tidy-select> Columns to include in the results, either by name or by numeric index. Use c() or list() to select with more than one expression and ?tidyselect::language for full details on the selection language.

id
: The name of a column in which to store the file path. This is useful when reading multiple input files and there is data in the file paths, such as the data collection date. If NULL (the default) no extra column is created.

locale
: The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

na
: Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.

comment
: A string used to identify comments. Any text after the comment characters will be silently ignored.

trim_ws
: Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?

skip
: Number of lines to skip before reading data.

n_max
: Maximum number of lines to read.

guess_max
: Maximum number of lines to use for guessing column types.

progress
: Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE.

name_repair
: Treatment of problematic column names:

- "minimal": No name repair or checks, beyond basic existence of names
- "unique": Make sure names are unique and not empty
- "check_unique": (default value), no name repair, but check they are unique
- "universal": Make the names unique and syntactic
- a function: apply custom name repair (e.g., .name_repair = make.names for names in the style of base R)
- A purrr-style anonymous function, see rlang::as_function()

This argument is passed on as repair to vctrs::vec_as_names(). See there for more details on these terms and the strategies used to enforce them.

num_threads — The number of processing threads to use for initial parsing and lazy reading of data. If your data contains newlines within fields the parser should automatically detect this and fall back to using one thread only. However if you know your file has newlines within quoted fields it is safest to set num_threads = 1 explicitly.

show_col_types — If FALSE, do not show the guessed column types. If TRUE always show the column types, even if they are supplied. If NULL (the default) only show the column types if they are not explicitly supplied by the col_types argument.

lazy — Read values lazily? By default the file is initially only indexed and the values are read lazily when accessed. Lazy reading is useful interactively, particularly if you are only interested in a subset of the full dataset. Note, lazy reading on windows will lock the file until all the data has been read from it, if you run into this issue set lazy = FALSE.

skip_empty_rows — Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.

col_names — Either NULL, or a character vector column names.

n — Number of lines the tokenizer will read to determine file structure. By default it is set to 100.

widths — Width of each field. Use NA as width of last field when reading a ragged fwf file.

start, end — Starting and ending (inclusive) positions of each field. Use NA as last end field when reading a ragged fwf file.

... — If the first element is a data frame, then it must have all numeric columns and either one or two rows. The column names are the variable names. The column values are the variable widths if a length one vector, and if length two, variable start and end positions. The elements of ... are used to construct a data frame with or or two rows as above.

## Second edition changes

Comments are no longer looked for anywhere in the file. They are now only ignored at the start of a line.

## See Also

read_table() to read fixed width files where each column is separated by whitespace.

## Examples

```
fwf_sample <- readr_example("fwf-sample.txt")
writeLines(read_lines(fwf_sample))

# You can specify column positions in several ways:
# 1. Guess based on position of empty columns
read_fwf(fwf_sample, fwf_empty(fwf_sample, col_names = c("first", "last", "state", "ssn")))
# 2. A vector of field widths
read_fwf(fwf_sample, fwf_widths(c(20, 10, 12), c("name", "state", "ssn")))
# 3. Paired vectors of start and end positions
read_fwf(fwf_sample, fwf_positions(c(1, 30), c(20, 42), c("name", "ssn")))
# 4. Named arguments with start and end positions
read_fwf(fwf_sample, fwf_cols(name = c(1, 20), ssn = c(30, 42)))
# 5. Named arguments with column widths
read_fwf(fwf_sample, fwf_cols(name = 20, state = 10, ssn = 12))
```

---

# read_lines *Read/write lines to/from a file*

---

## Description

`read_lines()` reads up to `n_max` lines from a file. New lines are not included in the output. `read_lines_raw()` produces a list of raw vectors, and is useful for handling data with unknown encoding. `write_lines()` takes a character vector or list of raw vectors, appending a new line after each entry.

## Usage

```
read_lines(
  file,
  skip = 0,
  skip_empty_rows = FALSE,
  n_max = Inf,
  locale = default_locale(),
  na = character(),
  lazy = TRUE,
  num_threads = readr_threads(),
  progress = show_progress()
)

read_lines_raw(
  file,
  skip = 0,
  n_max = -1L,
  num_threads = readr_threads(),
  progress = show_progress()
)

write_lines(
  x,
  file,
  sep = "\n",
```

```
  na = "NA",
  append = FALSE,
  num_threads = readr_threads(),
  path = deprecated()
)
```

## Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| skip | Number of lines to skip before reading data. |
| skip_empty_rows | Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns. |
| n_max | Number of lines to read. If n_max is -1, all lines in file will be read. |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| lazy | Read values lazily? By default the file is initially only indexed and the values are read lazily when accessed. Lazy reading is useful interactively, particularly if you are only interested in a subset of the full dataset. Note, lazy reading on windows will lock the file until all the data has been read from it, if you run into this issue set lazy = FALSE. |
| num_threads | The number of processing threads to use for initial parsing and lazy reading of data. If your data contains newlines within fields the parser should automatically detect this and fall back to using one thread only. However if you know your file has newlines within quoted fields it is safest to set num_threads = 1 explicitly. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE. |
| x | A character vector or list of raw vectors to write to disk. |
| sep | The line separator. Defaults to \\n, commonly used on POSIX systems like macOS and linux. For native windows (CRLF) separators use \\r\\n. |
| append | If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created. |
| path | **[Deprecated]**, use the file argument instead. |

## Value

`read_lines()`: A character vector with one element for each line. `read_lines_raw()`: A list containing a raw vector for each line.

`write_lines()` returns `x`, invisibly.

## Examples

```
read_lines(file.path(R.home("doc"), "AUTHORS"), n_max = 10)
read_lines_raw(file.path(R.home("doc"), "AUTHORS"), n_max = 10)

tmp <- tempfile()

write_lines(rownames(mtcars), tmp)
read_lines(tmp, lazy = FALSE)
read_file(tmp) # note trailing \n

write_lines(airquality$Ozone, tmp, na = "-1")
read_lines(tmp)
```

---

# read_log *Read common/combined log file into a tibble*

---

## Description

This is a fairly standard format for log files - it uses both quotes and square brackets for quoting, and there may be literal quotes embedded in a quoted string. The dash, "-", is used for missing values.

## Usage

```
read_log(
  file,
  col_names = FALSE,
  col_types = NULL,
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress()
)
```

## Arguments

| | |
|---|---|
| `file` | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of `clipboard()` will read from the system clipboard. |

col_names — Either TRUE, FALSE or a character vector of column names.

If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.

If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.

Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique, see name_repair to control how this is done.

col_types — One of NULL, a cols() specification, or a string. See vignette("readr") for more details.

If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to increase the guess_max or supply the correct types yourself.

Column specifications created by list() or cols() must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().

Alternatively, you can use a compact string representation where each character represents one column:

- c = character
- i = integer
- n = number
- d = double
- l = logical
- f = factor
- D = date
- T = date time
- t = time
- ? = guess
- _ or - = skip

By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, set show_col_types = FALSE or set 'options(readr.show_col_types = FALSE).

trim_ws — Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?

skip — Number of lines to skip before reading data. If comment is supplied any commented lines are ignored *after* skipping.

n_max — Maximum number of lines to read.

progress — Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE.

## Examples

```
read_log(readr_example("example.log"))
```

read_rds *Read/write RDS files.*

## Description

Consistent wrapper around `saveRDS()` and `readRDS()`. `write_rds()` does not compress by default as space is generally cheaper than time.

## Usage

```
read_rds(file, refhook = NULL)

write_rds(
  x,
  file,
  compress = c("none", "gz", "bz2", "xz"),
  version = 2,
  refhook = NULL,
  path = deprecated(),
  ...
)
```

## Arguments

| | |
|---|---|
| `file` | The file path to read from/write to. |
| `refhook` | A function to handle reference objects. |
| `x` | R object to write to serialise. |
| `compress` | Compression method to use: "none", "gz" ,"bz", or "xz". |
| `version` | Serialization format version to be used. The default value is 2 as it's compatible for R versions prior to 3.5.0. See `base::saveRDS()` for more details. |
| `path` | **[Deprecated]**, use the `file` argument instead. |
| `...` | Additional arguments to connection function. For example, control the space-time trade-off of different compression methods with `compression`. See `connections()` for more details. |

## Value

`write_rds()` returns `x`, invisibly.

## Examples

```
temp <- tempfile()
write_rds(mtcars, temp)
read_rds(temp)
## Not run:
write_rds(mtcars, "compressed_mtc.rds", "xz", compression = 9L)

## End(Not run)
```

`read_table` *Read whitespace-separated columns into a tibble*

## Description

`read_table()` is designed to read the type of textual data where each column is separated by one (or more) columns of space.

`read_table()` is like `read.table()`, it allows any number of whitespace characters between columns, and the lines can be of different lengths.

`spec_table()` returns the column specifications rather than a data frame.

## Usage

```
read_table(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  comment = "",
  skip_empty_rows = TRUE
)
```

## Arguments

| | |
|---|---|
| `file` | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of `clipboard()` will read from the system clipboard. |
| `col_names` | Either `TRUE`, `FALSE` or a character vector of column names.<br>If `TRUE`, the first row of the input will be used as the column names, and will not be included in the data frame. If `FALSE`, column names will be generated automatically: X1, X2, X3 etc.<br>If `col_names` is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.<br>Missing (`NA`) column names will generate a warning, and be filled in with dummy names `X1`, `X2` etc. Duplicate column names will generate a warning and be made unique, see `name_repair` to control how this is done. |

`col_types` One of `NULL`, a `cols()` specification, or a string. See `vignette("readr")` for more details.

If `NULL`, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to increase the `guess_max` or supply the correct types yourself.

Column specifications created by `list()` or `cols()` must contain one column specification for each column. If you only want to read a subset of the columns, use `cols_only()`.

Alternatively, you can use a compact string representation where each character represents one column:

- c = character
- i = integer
- n = number
- d = double
- l = logical
- f = factor
- D = date
- T = date time
- t = time
- ? = guess
- _ or - = skip

  By default, reading a file without a column specification will print a message showing what `readr` guessed they were. To remove this message, set `show_col_types = FALSE` or set 'options(readr.show_col_types = FALSE).

`locale` The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

`na` Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.

`skip` Number of lines to skip before reading data.

`n_max` Maximum number of lines to read.

`guess_max` Maximum number of lines to use for guessing column types.

`progress` Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.

`comment` A string used to identify comments. Any text after the comment characters will be silently ignored.

`skip_empty_rows`

Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.

## See Also

`read_fwf()` to read fixed width files where each column is not separated by whitespace. `read_fwf()` is also useful for reading tabular data with non-standard formatting.

## Examples

```
# One corner from http://www.masseyratings.com/cf/compare.htm
massey <- readr_example("massey-rating.txt")
cat(read_file(massey))
read_table(massey)

# Sample of 1978 fuel economy data from
# http://www.fueleconomy.gov/feg/epadata/78data.zip
epa <- readr_example("epa78.txt")
cat(read_file(epa))
read_table(epa, col_names = FALSE)
```

---

# should_show_types *Determine whether column types should be shown*

---

## Description

Column types are shown unless

- They are disabled by setting `options(readr.show_col_types = FALSE)`
- The column types are supplied with the `col_types` argument.

## Usage

```
should_show_types()
```

---

# show_progress *Determine whether progress bars should be shown*

---

## Description

Progress bars are shown *unless* one of the following is `TRUE`

- The bar is explicitly disabled by setting `options(readr.show_progress = FALSE)`
- The code is run in a non-interactive session (`interactive()` is `FALSE`).
- The code is run in an RStudio notebook chunk.
- The code is run by knitr / rmarkdown.

## Usage

```
show_progress()
```

spec_delim *Generate a column specification*

## Description

When printed, only the first 20 columns are printed by default. To override, set `options(readr.num_columns)` can be used to modify this (a value of 0 turns off printing).

## Usage

```
spec_delim(
  file,
  delim = NULL,
  quote = "\"",
  escape_backslash = FALSE,
  escape_double = TRUE,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = 0,
  guess_max = 1000,
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

spec_csv(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = 0,
  guess_max = 1000,
```

```
  name_repair = "unique",
  num_threads = readr_threads(),
  progress = show_progress(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

spec_csv2(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = 0,
  guess_max = 1000,
  progress = show_progress(),
  name_repair = "unique",
  num_threads = readr_threads(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
)

spec_tsv(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = 0,
  guess_max = 1000,
  progress = show_progress(),
  name_repair = "unique",
  num_threads = readr_threads(),
  show_col_types = should_show_types(),
  skip_empty_rows = TRUE,
  lazy = TRUE
```

```
)

spec_table(
  file,
  col_names = TRUE,
  col_types = list(),
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = 0,
  guess_max = 1000,
  progress = show_progress(),
  comment = "",
  skip_empty_rows = TRUE
)
```

## Arguments

| | |
|---|---|
| file | Either a path to a file, a connection, or literal data (either a single string or a raw vector).<br>Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.<br>Literal data is most useful for examples and tests. To be recognised as a path, it must be wrapped with I(), be a string containing at least one new line, or be a vector containing at least one string with a new line.<br>Using a value of clipboard() will read from the system clipboard. |
| delim | Single character used to separate fields within a record. |
| quote | Single character used to quote strings. |
| escape_backslash | Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \\n. |
| escape_double | Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, \". |
| col_names | Either TRUE, FALSE or a character vector of column names.<br>If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.<br>If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.<br>Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique, see name_repair to control how this is done. |
| col_types | One of NULL, a cols() specification, or a string. See vignette("readr") for more details.<br>If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to increase the guess_max or supply the correct types yourself. |

Column specifications created by list() or cols() must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().

Alternatively, you can use a compact string representation where each character represents one column:

- c = character
- i = integer
- n = number
- d = double
- l = logical
- f = factor
- D = date
- T = date time
- t = time
- ? = guess
- _ or - = skip

By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, set show_col_types = FALSE or set 'options(readr.show_col_types = FALSE).

col_select <tidy-select> Columns to include in the results, either by name or by numeric index. Use c() or list() to select with more than one expression and ?tidyselect::language for full details on the selection language.

id The name of a column in which to store the file path. This is useful when reading multiple input files and there is data in the file paths, such as the data collection date. If NULL (the default) no extra column is created.

locale The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

na Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.

quoted_na Should missing values inside quotes be treated as missing values (the default) or strings.

comment A string used to identify comments. Any text after the comment characters will be silently ignored.

trim_ws Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?

skip Number of lines to skip before reading data. If comment is supplied any commented lines are ignored *after* skipping.

n_max Maximum number of lines to read.

guess_max Maximum number of lines to use for guessing column types.

name_repair Treatment of problematic column names:

- "minimal": No name repair or checks, beyond basic existence of names
- "unique": Make sure names are unique and not empty
- "check_unique": (default value), no name repair, but check they are unique
- "universal": Make the names unique and syntactic

- a function: apply custom name repair (e.g., `.name_repair = make.names` for names in the style of base R)
- A purrr-style anonymous function, see `rlang::as_function()`

This argument is passed on as `repair` to `vctrs::vec_as_names()`. See there for more details on these terms and the strategies used to enforce them.

`num_threads` The number of processing threads to use for initial parsing and lazy reading of data. If your data contains newlines within fields the parser should automatically detect this and fall back to using one thread only. However if you know your file has newlines within quoted fields it is safest to set `num_threads = 1` explicitly.

`progress` Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.

`show_col_types` If `FALSE`, do not show the guessed column types. If `TRUE` always show the column types, even if they are supplied. If `NULL` (the default) only show the column types if they are not explicitly supplied by the `col_types` argument.

`skip_empty_rows`

Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.

`lazy` Read values lazily? By default the file is initially only indexed and the values are read lazily when accessed. Lazy reading is useful interactively, particularly if you are only interested in a subset of the full dataset. Note, lazy reading on windows will lock the file until all the data has been read from it, if you run into this issue set `lazy = FALSE`.

## Value

The `col_spec` generated for the file.

## Examples

```
# Input sources -------------------------------------------------------------
# Retrieve specs from a path
spec_csv(system.file("extdata/mtcars.csv", package = "readr"))
spec_csv(system.file("extdata/mtcars.csv.zip", package = "readr"))

# Or directly from a string (must contain a newline)
spec_csv(I("x,y\n1,2\n3,4"))

# Column types --------------------------------------------------------------
# By default, readr guesses the columns types, looking at the first 1000 rows.
# You can specify the number of rows used with guess_max.
spec_csv(system.file("extdata/mtcars.csv", package = "readr"), guess_max = 20)
```

---

# type_convert *Re-convert character columns in existing data frame*

---

## Description

This is useful if you need to do some manual munging - you can read the columns in as character, clean it up with (e.g.) regular expressions and then let readr take another stab at parsing it. The name is a homage to the base `utils::type.convert()`.

## Usage

```
type_convert(
  df,
  col_types = NULL,
  na = c("", "NA"),
  trim_ws = TRUE,
  locale = default_locale()
)
```

## Arguments

| | |
|---|---|
| df | A data frame. |
| col_types | One of NULL, a cols() specification, or a string. See vignette("readr") for more details.<br>If NULL, column types will be imputed using all rows. |
| na | Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values. |
| trim_ws | Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it? |
| locale | The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names. |

## Note

type_convert() removes a 'spec' attribute, because it likely modifies the column data types. (see spec() for more information about column specifications).

## Examples

```
df <- data.frame(
  x = as.character(runif(10)),
  y = as.character(sample(10)),
  stringsAsFactors = FALSE
)
str(df)
str(type_convert(df))

df <- data.frame(x = c("NA", "10"), stringsAsFactors = FALSE)
str(type_convert(df))

# Type convert can be used to infer types from an entire dataset

# first read the data as character
data <- read_csv(readr_example("mtcars.csv"),
  col_types = list(.default = col_character())
)
str(data)
# Then convert it with type_convert
type_convert(data)
```

## with_edition *Temporarily change the active readr edition*

### Description

`with_edition()` allows you to change the active edition of readr for a given block of code. `local_edition()` allows you to change the active edition of readr until the end of the current function or file.

### Usage

```
with_edition(edition, code)

local_edition(edition, env = parent.frame())
```

### Arguments

| | |
|---|---|
| `edition` | Should be a single integer. |
| `code` | Code to run with the changed edition. |
| `env` | Environment that controls scope of changes. For expert use only. |

## write_delim *Write a data frame to a delimited file*

### Description

The write_*() family of functions are an improvement to analogous function such as `write.csv()` because they are approximately twice as fast. Unlike `write.csv()`, these functions do not include row names as a column in the written file. A generic function, `output_column()`, is applied to each variable to coerce columns to suitable output.

### Usage

```
write_delim(
  x,
  file,
  delim = " ",
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)

write_csv(
```

```
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)

write_csv2(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = c("needed", "all", "none"),
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)

write_excel_csv(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  delim = ",",
  quote = "all",
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)

write_excel_csv2(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  delim = ";",
```

```
  quote = "all",
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)

write_tsv(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote = "none",
  escape = c("double", "backslash", "none"),
  eol = "\n",
  num_threads = readr_threads(),
  progress = show_progress(),
  path = deprecated(),
  quote_escape = deprecated()
)
```

## Arguments

| | |
|---|---|
| x | A data frame or tibble to write to disk. |
| file | File or connection to write to. |
| delim | Delimiter used to separate values. Defaults to " " for write_delim(), "," for write_excel_csv() and ";" for write_excel_csv2(). Must be a single character. |
| na | String used for missing values. Defaults to NA. Missing values will never be quoted; strings with the same value as na will always be quoted. |
| append | If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created. |
| col_names | If FALSE, column names will not be included at the top of the file. If TRUE, column names will be included. If not specified, col_names will take the opposite value given to append. |
| quote | How to handle fields which contain characters that need to be quoted.<br>• needed - Only quote fields which need them.<br>• all - Quote all fields.<br>• none - Never quote fields. |
| escape | The type of escape to use when quotes are in the data.<br>• double - quotes are escaped by doubling them.<br>• backslash - quotes are escaped by a preceding backslash.<br>• none - quotes are not escaped. |
| eol | The end of line character to use. Most commonly either "\n" for Unix style newlines, or "\r\n" for Windows style newlines. |

| | |
|---|---|
| num_threads | Number of threads to use when reading and materializing vectors. If your data contains newlines within fields the parser will automatically be forced to use a single thread only. |
| progress | Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE. |
| path | **[Deprecated]**, use the file argument instead. |
| quote_escape | **[Deprecated]**, use the escape argument instead. |

## Value

write_*() returns the input x invisibly.

## Output

Factors are coerced to character. Doubles are formatted to a decimal string using the grisu3 algorithm. POSIXct values are formatted as ISO8601 with a UTC timezone *Note:* POSIXct *objects in local or non-UTC timezones will be converted to UTC time before writing.*

All columns are encoded as UTF-8. write_excel_csv() and write_excel_csv2() also include a UTF-8 Byte order mark which indicates to Excel the csv is UTF-8 encoded.

write_excel_csv2() and write_csv2 were created to allow users with different locale settings to save .csv files using their default settings (e.g. ; as the column separator and , as the decimal separator). This is common in some European countries.

Values are only quoted if they contain a comma, quote or newline.

The write_*() functions will automatically compress outputs if an appropriate extension is given. Three extensions are currently supported: .gz for gzip compression, .bz2 for bzip2 compression and .xz for lzma compression. See the examples for more information.

## References

Florian Loitsch, Printing Floating-Point Numbers Quickly and Accurately with Integers, PLDI '10, http://www.cs.tufts.edu/~nr/cs257/archive/florian-loitsch/printf.pdf

## Examples

```
# If only a file name is specified, write_()* will write
# the file to the current working directory.
write_csv(mtcars, "mtcars.csv")
write_tsv(mtcars, "mtcars.tsv")

# If you add an extension to the file name, write_()* will
# automatically compress the output.
write_tsv(mtcars, "mtcars.tsv.gz")
write_tsv(mtcars, "mtcars.tsv.bz2")
write_tsv(mtcars, "mtcars.tsv.xz")
```

# Index