Package 'readr'

May 9, 2026

Title Read Rectangular Text Data

Version 2.2.0

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 (>= 4.1)

Imports cli, clipr, crayon, glue, hms (>= 0.4.1), lifecycle, methods, R6, rlang, tibble, utils, vroom (>= 1.7.0), withr

Suggests covr, curl, datasets, knitr, rmarkdown, spelling, stringi, testthat (>= 3.2.0), tzdb (>= 0.1.1), waldo, xml2

LinkingTo cpp11, tzdb (>= 0.1.1)

VignetteBuilder knitr

Config/Needs/website tidyverse, tidyverse/tidytemplate

Config/testthat/edition 3

Config/testthat/parallel false

Config/usethis/last-upkeep 2025-11-14

Encoding UTF-8

Language en-US

RoxygenNote 7.3.3

NeedsCompilation yes

Author Hadley Wickham [aut],
Jim Hester [aut],
Romain Francois [ctb],
Jennifer Bryan [aut, cre] (ORCID:
<https://orcid.org/0000-0002-6983-2759>),

Shelby Bearrows [ctb],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>),
https://github.com/mandreyel/ [cph] (mio library),
Jukka Jylänki [ctb, cph] (grisu3 implementation),
Mikkel Jørgensen [ctb, cph] (grisu3 implementation)

Maintainer Jennifer Bryan <jenny@posit.co>

Repository CRAN

Date/Publication 2026-02-19 18:10:02 UTC

Contents

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()`, `cols_condense()`, `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). 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. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with `I()`. 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"` 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.

Examples

```
edition_get()
```

`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"
)

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

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

format_tsv(
  x,
```

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

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. • needed - Values are only quoted if needed: if they contain a delimiter, quote, or newline. • all - Quote all fields. • 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.

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. Use "" to use the system default time zone, but beware that this will not be reproducible across systems. 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")

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

---

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(), cols_condense(), 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(). 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(), cols_condense(), 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 generates a warning if `levels` have been specified and some elements of `x` are not found in those `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 of the allowed levels. When `levels = NULL` (the default), `levels` are discovered from the unique values of `x`, in the order in which they appear 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 TRUE and x contains at least one NA, then NA is included in the levels of the constructed factor.
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(), cols_condense(), parse_datetime(), parse_guess(), parse_logical(), parse_number(), parse_vector()

Examples

```
# discover the levels from the data
parse_factor(c("a", "b"))
parse_factor(c("a", "b", "-99"))
parse_factor(c("a", "b", "-99"), na = c("", "NA", "-99"))
parse_factor(c("a", "b", "-99"), na = c("", "NA", "-99"), include_na = FALSE)

# provide the levels explicitly
parse_factor(c("a", "b"), levels = letters[1:5])

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

# base::factor() silently converts elements that do not match any levels to
# NA
factor(x, levels = animals)

# parse_factor() generates same factor as base::factor() but throws a warning
# and reports problems
parse_factor(x, levels = animals)
```

---

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()`, `cols_condense()`, `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 parses the first number it finds, dropping any non-numeric characters before the first number and all characters 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()`, `cols_condense()`, `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("t1000t1000") ## only parses first number found

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`	A 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

```
read_builtin("mtcars", "datasets")
```

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 = deprecated(),
  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 = should_read_lazy()
)

read_csv(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)

read_csv2(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
```

```
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)

read_tsv(
  file,
  col_names = TRUE,
  col_types = NULL,
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)
```

Arguments

`file`	Either a path to a file, a connection, or literal data (either a single string or a raw vector). `file` can also be a character vector containing multiple filepaths or a list containing multiple connections. Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically decompressed. Files starting with `http://`, `https://`, `ftp://`, or `ftps://` will be automatically downloaded. Remote compressed files (`.gz`, `.bz2`, `.xz`, `.zip`) will be automatically downloaded and decompressed.

Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with `I()`.

`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.

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 `...1`, `...2` 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 inferred from `guess_max` rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase `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 — Columns to include in the results. You can use the same mini-language as dplyr::select() to refer to the columns by name. Use c() to use more than one selection expression. Although this usage is less common, col_select also accepts a numeric column index. See ?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 — **[Deprecated]** Should missing values inside quotes be treated as missing values (the default) or strings. This argument is deprecated and only works when using the legacy first edition parser. See with_edition() for more.

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. Will never use more than the number of lines read. See vignette("column-types", package = "readr") for more details.

name_repair — Handling of column names. The default behaviour is to ensure column names are "unique". Various repair strategies are supported:

- "minimal": No name repair or checks, beyond basic existence of names.
- "unique" (default value): Make sure names are unique and not empty.
- "check_unique": No name repair, but check they are unique.
- "unique_quiet": Repair with the unique strategy, quietly.
- "universal": Make the names unique and syntactic.
- "universal_quiet": Repair with the universal strategy, quietly.
- 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, this is `FALSE`, because there are special considerations when reading a file lazily that have tripped up some users. Specifically, things get tricky when reading and then writing back into the same file. But, in general, lazy reading (`lazy = TRUE`) has many benefits, especially for interactive use and when your downstream work only involves a subset of the rows or columns.

  Learn more in `should_read_lazy()` and in the documentation for the `altrep` argument of `vroom::vroom()`.

Value

A `tibble::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/main/inst/extdata/mtcars.csv")

## End(Not run)

# Read from multiple file paths at once
continents <- c("africa", "americas", "asia", "europe", "oceania")
filepaths <- vapply(
  paste0("mini-gapminder-", continents, ".csv"),
  FUN = readr_example,
  FUN.VALUE = character(1)
)
read_csv(filepaths, id = "file")

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

# Column selection----------------------------------------------------------
```

```
# Pass column names or indexes directly to select them
read_csv(readr_example("chickens.csv"), col_select = c(chicken, eggs_laid))
read_csv(readr_example("chickens.csv"), col_select = c(1, 3:4))

# Or use the selection helpers
read_csv(
  readr_example("chickens.csv"),
  col_select = c(starts_with("c"), last_col())
)

# You can also rename specific columns
read_csv(
  readr_example("chickens.csv"),
  col_select = c(egg_yield = eggs_laid, everything())
)

# Column types ------------------------------------------------------------
# By default, readr guesses the columns types, looking at `guess_max` 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)

# Column names ------------------------------------------------------------
# By default, readr duplicate name repair is noisy
read_csv(I("x,x\n1,2\n3,4"))

# Same default repair strategy, but quiet
read_csv(I("x,x\n1,2\n3,4"), name_repair = "unique_quiet")

# There's also a global option that controls verbosity of name repair
withr::with_options(
  list(rlib_name_repair_verbosity = "quiet"),
  read_csv(I("x,x\n1,2\n3,4"))
)

# Or use "minimal" to turn off name repair
read_csv(I("x,x\n1,2\n3,4"), name_repair = "minimal")

# 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)
```

Arguments

file	Either a path to a file, a connection, or literal data (either a single string or a raw vector). 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. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with I(). 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.

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

Fixed-width files store tabular data with each field occupying a specific range of character positions in every line. Once the fields are identified, converting them to the appropriate R types works just like for delimited files. The unique challenge with fixed-width files is describing where each field begins and ends. **readr** tries to ease this pain by offering a few different ways to specify the field structure:

- `fwf_empty()` - Guesses based on the positions of empty columns. This is the default. (Note that `fwf_empty()` returns 0-based positions, for internal use.)
- `fwf_widths()` - Supply the widths of the columns.
- `fwf_positions()` - Supply paired vectors of start and end positions. These are interpreted as 1-based positions, so are off-by-one compared to the output of `fwf_empty()`.
- `fwf_cols()` - Supply named arguments of paired start and end positions or column widths.

Note: `fwf_empty()` cannot work with a connection or with any of the input types that involve a connection internally, which includes remote and compressed files. The reason is that this would necessitate reading from the connection twice. In these cases, you'll have to either provide the field structure explicitly with another `fwf_*()` function or download (and decompress, if relevant) the file first.

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 = should_read_lazy(),
  skip_empty_rows = TRUE
)

fwf_empty(
  file,
  skip = 0,
  skip_empty_rows = deprecated(),
  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). 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. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with `I()`. Using a value of `clipboard()` will read from the system clipboard.
`col_positions`	Column positions, as created by `fwf_empty()`, `fwf_widths()`, `fwf_positions()`, or `fwf_cols()`. 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 inferred from `guess_max` rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase `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` Columns to include in the results. You can use the same mini-language as `dplyr::select()` to refer to the columns by name. Use `c()` to use more than one selection expression. Although this usage is less common, `col_select` also accepts a numeric column index. See `?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. Will never use more than the number of lines read. See `vignette("column-types", package = "readr")` for more details.

`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` Handling of column names. The default behaviour is to ensure column names are `"unique"`. Various repair strategies are supported:

- `"minimal"`: No name repair or checks, beyond basic existence of names.

- "unique" (default value): Make sure names are unique and not empty.
- "check_unique": No name repair, but check they are unique.
- "unique_quiet": Repair with the unique strategy, quietly.
- "universal": Make the names unique and syntactic.
- "universal_quiet": Repair with the universal strategy, quietly.
- 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, this is FALSE, because there are special considerations when reading a file lazily that have tripped up some users. Specifically, things get tricky when reading and then writing back into the same file. But, in general, lazy reading (lazy = TRUE) has many benefits, especially for interactive use and when your downstream work only involves a subset of the rows or columns.

Learn more in should_read_lazy() and in the documentation for the altrep argument of vroom::vroom().

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 the width of the last field when reading a ragged fixed-width file.

start, end — Starting and ending (inclusive) positions of each field. **Positions are 1-based**: the first character in a line is at position 1. Use NA as the last value of end when reading a ragged fixed-width 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.

Details

Here's a enhanced example using the contents of the file accessed via `readr_example("fwf-sample.txt")`.

```
         1         2         3         4
12345678901234567890123456789012345678901234567890
[     name 20      ][state 10][  ssn 12  ]
John Smith          WA        418-Y11-4111
Mary Hartford       CA        319-Z19-4341
Evan Nolan          IL        219-532-c301
```

Here are some valid field specifications for the above (they aren't all equivalent! but they are all valid):

```
fwf_widths(c(20, 10, 12), c("name", "state", "ssn"))
fwf_positions(c(1, 30), c(20, 42), c("name", "ssn"))
fwf_cols(state = c(21, 30), last = c(6, 20), first = c(1, 4), ssn = c(31, 42))
fwf_cols(name = c(1, 20), ssn = c(30, 42))
fwf_cols(name = 20, state = 10, ssn = 12)
```

Second edition changes

Comments are now only ignored if they appear at the start of a line. Comments elsewhere in a line are no longer treated specially.

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 = should_read_lazy(),
  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()
)
```

Arguments

file	Either a path to a file, a connection, or literal data (either a single string or a raw vector).

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.

Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with I().

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, this is FALSE, because there are special considerations when reading a file lazily that have tripped up some users. Specifically, things get tricky when reading and then writing back into the same file. But, in general, lazy reading (lazy = TRUE) has many benefits, especially for interactive use and when your downstream work only involves a subset of the rows or columns.

Learn more in should_read_lazy() and in the documentation for the altrep argument of vroom::vroom().

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.

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,
  show_col_types = should_show_types(),
  progress = show_progress()
)
```

Arguments

file	Either a path to a file, a connection, or literal data (either a single string or a raw vector). 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. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with I(). 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 ...1, ...2 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 inferred from guess_max rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase 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.

n_max
: Maximum number of lines to read.

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.

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,
  text = FALSE,
  ...
)
```

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.
`text`	If `TRUE` a text representation is used, otherwise a binary representation is used.
`...`	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 = "",
  show_col_types = should_show_types(),
  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). 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. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with `I()`.

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 ...1, ...2 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 inferred from guess_max rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase 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. Will never use more than the number of lines read. See `vignette("column-types", package = "readr")` for more details.

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.

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.

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

```
ws <- readr_example("whitespace-sample.txt")
writeLines(read_lines(ws))
read_table(ws)
```

---

should_read_lazy *Determine whether to read a file lazily*

Description

This function consults the option `readr.read_lazy` to figure out whether to do lazy reading or not. If the option is unset, the default is `FALSE`, meaning readr will read files eagerly, not lazily. If you want to use this option to express a preference for lazy reading, do this:

```
options(readr.read_lazy = TRUE)
```

Typically, one would use the option to control lazy reading at the session, file, or user level. The `lazy` argument of functions like `read_csv()` can be used to control laziness in an individual call.

Usage

```
should_read_lazy()
```

See Also

The blog post "Eager vs lazy reading in readr 2.1.0" explains the benefits (and downsides) of lazy reading.

should_show_types *Determine whether column types should be shown*

Description

Wrapper around `getOption("readr.show_col_types")` that implements some fall back logic if the option is unset. This returns:

- `TRUE` if the option is set to `TRUE`
- `FALSE` if the option is set to `FALSE`
- `FALSE` if the option is unset and we appear to be running tests
- `NULL` otherwise, in which case the caller determines whether to show column types based on context, e.g. whether `show_col_types` or actual `col_types` were explicitly specified

Usage

```
should_show_types()
```

show_progress *Determine whether progress bars should be shown*

Description

By default, readr shows progress bars. However, progress reporting is suppressed if any of the following conditions hold:

- The bar is explicitly disabled by setting `options(readr.show_progress = FALSE)`.
- The code is run in a non-interactive session, as determined by `rlang::is_interactive()`.
- The code is run in an RStudio notebook chunk, as determined by `getOption("rstudio.notebook.executing")`.

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 = deprecated(),
  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 = should_read_lazy()
)

spec_csv(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)

spec_csv2(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)

spec_tsv(
  file,
  col_names = TRUE,
  col_types = list(),
  col_select = NULL,
  id = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = deprecated(),
  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 = should_read_lazy()
)

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 = "",
  show_col_types = should_show_types(),
  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). `file` can also be a character vector containing multiple filepaths or a list containing multiple connections. Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically decompressed. Files starting with `http://`, `https://`, `ftp://`, or `ftps://` will be automatically downloaded. Remote compressed files (`.gz`, `.bz2`, `.xz`, `.zip`) will be automatically downloaded and decompressed. Literal data is most useful for examples and tests. To be recognised as literal data, wrap the input with `I()`.
`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. 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 `...1`, `...2` 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 inferred from `guess_max` rows of the input, interspersed throughout the file. This is convenient (and fast), but not robust. If the guessed types are wrong, you'll need to increase `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` Columns to include in the results. You can use the same mini-language as `dplyr::select()` to refer to the columns by name. Use `c()` to use more than one selection expression. Although this usage is less common, `col_select` also accepts a numeric column index. See `?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 **[Deprecated]** Should missing values inside quotes be treated as missing values (the default) or strings. This argument is deprecated and only works when using the legacy first edition parser. See `with_edition()` for more.

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. Will never use more than the number of lines read. See `vignette("column-types", package = "readr")` for more details.

name_repair Handling of column names. The default behaviour is to ensure column names are `"unique"`. Various repair strategies are supported:

- `"minimal"`: No name repair or checks, beyond basic existence of names.
- `"unique"` (default value): Make sure names are unique and not empty.
- `"check_unique"`: No name repair, but check they are `unique`.
- `"unique_quiet"`: Repair with the `unique` strategy, quietly.
- `"universal"`: Make the names `unique` and syntactic.
- `"universal_quiet"`: Repair with the `universal` strategy, quietly.
- 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, this is `FALSE`, because there are special considerations when reading a file lazily that have tripped up some users. Specifically, things get tricky when reading and then writing back into the same file. But,

in general, lazy reading (`lazy = TRUE`) has many benefits, especially for interactive use and when your downstream work only involves a subset of the rows or columns.

Learn more in `should_read_lazy()` and in the documentation for the `altrep` argument of `vroom::vroom()`.

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 1000 rows
# throughout the file.
# 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(),
  guess_integer = FALSE
)
```

Arguments

df	A data frame.
col_types	One of NULL, a cols() specification, or a string. See vignette("readr") for more details. 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.
guess_integer	If TRUE, guess integer types for whole numbers, if FALSE guess numeric type for all numbers.

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, such as 1 or 2.
`code`	Code to run with the changed edition.
`env`	Environment that controls scope of changes. For expert use only.

Examples

```
with_edition(1, edition_get())
with_edition(2, edition_get())

# readr 1e and 2e behave differently when input rows have different number
# number of fields
with_edition(1, read_csv("1,2\n3,4,5", col_names = c("X", "Y", "Z")))
with_edition(2, read_csv("1,2\n3,4,5", col_names = c("X", "Y", "Z")))

# local_edition() applies in a specific scope, for example, inside a function
read_csv_1e <- function(...) {
  local_edition(1)
  read_csv(...)
}
read_csv("1,2\n3,4,5", col_names = c("X", "Y", "Z"))      # 2e behaviour
read_csv_1e("1,2\n3,4,5", col_names = c("X", "Y", "Z"))   # 1e behaviour
read_csv("1,2\n3,4,5", col_names = c("X", "Y", "Z"))      # 2e behaviour
```

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()
)

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()
)

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()
)

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()
)

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()
)

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()
)
```

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.

- `needed` - Values are only quoted if needed: if they contain a delimiter, quote, or newline.
- `all` - Quote all fields.
- `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.

`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 executing in an RStudio notebook chunk. The display of the progress bar can be disabled by setting the environment variable `VROOM_SHOW_PROGRESS` to `"false"`.

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