Package 'readr'

March 22, 2017

Version 1.1.0

Title Read Rectangular Text Data

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.

Encoding UTF-8

Depends R (>= 3.0.2)

LinkingTo Rcpp, BH

Imports Rcpp (>= 0.12.0.5), tibble, hms, R6

Suggests curl, testthat, knitr, rmarkdown, stringi, covr

License GPL (>= 2) | file LICENSE

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

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

VignetteBuilder knitr

RoxygenNote 6.0.1

NeedsCompilation yes

Author Hadley Wickham [aut],
Jim Hester [aut, cre],
Romain Francois [aut],
R Core Team [ctb] (Date time code adapted from R),
RStudio [cph, fnd],
Jukka Jylänki [ctb, cph] (grisu3 implementation),
Mikkel Jørgensen [ctb, cph] (grisu3 implementation)

Maintainer Jim Hester <james.hester@rstudio.com>

Repository CRAN

Date/Publication 2017-03-22 19:24:23 UTC

R topics documented:

---

cols	*Create column specification*

Description

Create column specification

Usage

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

cols_only(...)
```

Arguments

`...`	Either column objects created by `col_*()`, or their abbreviated character names. 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.

Examples

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

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

---

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.

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: [parse_datetime](), [parse_factor](), [parse_guess](), [parse_logical](), [parse_number]()

---

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. It must contain at least one new line to be recognised as data (instead of a path).

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 languge and a region, e.g. `"en_US"` for American English. See `date_names_locales()` for a complete list of available locales.

Examples

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

---

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)

format_csv(x, na = "NA", append = FALSE, col_names = !append)

format_tsv(x, na = "NA", append = FALSE, col_names = !append)
```

Arguments

x	A data frame to write to disk
delim	Delimiter used to separate values. Defaults to " ". 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 file does not exist a new file is created.
col_names	Write columns names at the top of the file?

Value

A string.

Output

Factors are coerced to character. Doubles are formatted using the grisu3 algorithm. POSIXct's are formatted as ISO8601.

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

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

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

---

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. Dec-
                    imal mark can only be , or ..

tz   Default tz. This is used both for input (if the time zone isn't present in indi-
                    vidual 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")

# 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())

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

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

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

col_logical()

col_integer()

col_double()

col_character()
```

Arguments

x	Character vector of values to parse.
na	Character vector of strings to use for 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.

See Also

Other parsers: `col_skip`, `parse_datetime`, `parse_factor`, `parse_guess`, `parse_number`

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())

parse_date(x, format = "", na = c("", "NA"), locale = default_locale())

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

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 use for 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()](link) to create your own locale that
            controls things like the default time zone, encoding, decimal mark, big mark,
            and day/month names.

Value

A [POSIXct()](link) vector with tzone attribute set to tz. Elements that could not be parsed (or did not
generate valid dates) will bes 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()](link). 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)

- Hour: "%H" or "%I", use I (and not H) with AM/PM.

- 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.

- 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, parse_factor, parse_guess, parse_logical, parse_number

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, ordered = FALSE, na = c("", "NA"),
  locale = default_locale(), include_na = TRUE)

col_factor(levels, 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 use for 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?

See Also

Other parsers: col_skip, parse_datetime, parse_guess, parse_logical, parse_number

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())

col_guess()

guess_parser(x, locale = default_locale())
```

Arguments

x   Character vector of values to parse.

na   Character vector of strings to use for 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.

See Also

Other parsers: col_skip, parse_datetime, parse_factor, parse_logical, parse_number

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())

col_number()
```

Arguments

x   Character vector of values to parse.

na   Character vector of strings to use for 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.

See Also

Other parsers: `col_skip`, `parse_datetime`, `parse_factor`, `parse_guess`, `parse_logical`

Examples

```
parse_number("$1000")
parse_number("1,234,567.78")
```

---

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)

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)
```

---

read_delim   *Read a delimited file (including csv & tsv) into a tibble*

Description

read_csv() and read_tsv() are special cases of the 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 separators, instead of ,. This is common in European countries which use , as the decimal separator.

Usage

```
read_delim(file, delim, quote = "\"", escape_backslash = FALSE,
  escape_double = TRUE, col_names = TRUE, col_types = 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), progress = show_progress())

read_csv(file, col_names = TRUE, col_types = 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())

read_csv2(file, col_names = TRUE, col_types = 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())

read_tsv(file, col_names = TRUE, col_types = 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())
```

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. It must contain at least one new line to be recognised as data (instead of a path).
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 X1, X2 etc. Duplicate column names will generate a warning and be made unique with a numeric prefix.

col_types   One of NULL, a [cols()](#) specification, or a string. See vignette("column-types") 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 supply the correct types yourself.

> If a column specification created by [cols()](#), it 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, D = date, T = date time, t = time, ? = guess, or _/- to skip the column.

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 use for 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 be trimmed from each field before parsing it?

skip   Number of lines to skip before reading data.

n_max   Maximum number of records to read.

guess_max   Maximum number of records 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 display is updated every 50,000 values

and will only display if estimated reading time is 5 seconds or more. The auto-
matic progress bar can be disabled by setting option `readr.show_progress` to
`FALSE`.

Value

A data frame. If there are parsing problems, a warning tells you how many, and you can retrieve the
details with [problems]().

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"))
read_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")

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

# Column types --------------------------------------------------------------
# By default, readr guesses the columns types, looking at the first 100 rows.
# You can override with a compact specification:
read_csv("x,y\n1,2\n3,4", col_types = "dc")

# Or with a list of column types:
read_csv("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("x\n1\n2\nb", col_types = list(col_double()))
y
problems(y)

# File types ----------------------------------------------------------------
read_csv("a,b\n1.0,2.0")
read_csv2("a;b\n1,0;2,0")
read_tsv("a\tb\n1.0\t2.0")
read_delim("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, path, 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. It must contain at least one new line to be recognised as data (instead of a path).
`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 data frame to write to disk
`path`	Path or connection to write to.
`append`	If `FALSE`, will overwrite existing file. If `TRUE`, will append to existing file. In both cases, if 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(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.

Usage

```
read_fwf(file, col_positions, col_types = NULL, locale = default_locale(),
  na = c("", "NA"), comment = "", skip = 0, n_max = Inf,
  guess_max = min(n_max, 1000), progress = show_progress())

fwf_empty(file, skip = 0, 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. It must contain at least one new line to be recognised as data (instead of a path).
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("column-types")` 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 supply the correct types yourself.
	If a column specification created by [cols()](), it 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, D = date, T = date time, t = time, ? = guess, or _/- to skip the column.

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 use for 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.

skip   Number of lines to skip before reading data.

n_max   Maximum number of records to read.

guess_max   Maximum number of records 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 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`.

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, and the column values are the variable widths if a length one vector, and variable start and end positions. Otherwise, the elements of ... are used to construct a data frame with or or two rows as above.

See Also

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

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
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(10, 42), c("name", "ssn")))
# 4. Named arguments with start and end positions
read_fwf(fwf_sample, fwf_cols(name = c(1, 10), 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, n_max = -1L, locale = default_locale(),
  na = character(), progress = show_progress())

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

write_lines(x, path, na = "NA", 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. It must contain at least one new line to be recognised as data (instead of a path).

skip
: Number of lines to skip before reading data.

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 use for missing values. Set this option to character() to indicate no missing values.

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`.
x	A data frame to write to disk
path	Path or connection to write to.
append	If `FALSE`, will overwrite existing file. If `TRUE`, will append to existing file. In both cases, if 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)
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, 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).
	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. It must contain at least one new line to be recognised as data (instead of a path).
`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 with a numeric prefix.
`col_types`	One of `NULL`, a [`cols()`](#) specification, or a string. See `vignette("column-types")` 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 supply the correct types yourself.
	If a column specification created by [`cols()`](#), it 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, D = date, T = date time, t = time, ? = guess, or _/- to skip the column.
`skip`	Number of lines to skip before reading data.
`n_max`	Maximum number of records to read.
`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`.

Examples

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

read_table	*Read whitespace-separated columns into a tibble*

Description

read_table() and read_table2() are designed to read the type of textual data where each column is #' separate by one (or more) columns of space.

read_table2() is like [read.table()](), it allows any number of whitespace characters between columns, and the lines can be of different lengths.

read_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.

spec_table() and spec_table2() return 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 = "")

read_table2(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 = "")
```

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. It must contain at least one new line to be recognised as data (instead of a path).
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 with a numeric prefix.

col_types	One of NULL, a [cols()](#) specification, or a string. See vignette("column-types") 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 supply the correct types yourself.
	If a column specification created by [cols()](#), it 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, D = date, T = date time, t = time, ? = guess, or _/- to skip the column.
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 use for 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 records to read.
guess_max	Maximum number of records 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 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.
comment	A string used to identify comments. Any text after the comment characters will be silently ignored.

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)
```

| 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, quote = "\"", escape_backslash = FALSE,
  escape_double = TRUE, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  comment = "", trim_ws = FALSE, skip = 0, n_max = 0,
  guess_max = 1000, progress = show_progress())

spec_csv(file, col_names = TRUE, col_types = 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())

spec_csv2(file, col_names = TRUE, col_types = 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())

spec_tsv(file, col_names = TRUE, col_types = 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())

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

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. It must contain at least one new line to be recognised as data (instead of a path).
`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 `X1`, `X2` etc. Duplicate column names will generate a warning and be made unique with a numeric prefix.

col_types   One of `NULL`, a [`cols()`](#) specification, or a string. See `vignette("column-types")` 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 supply the correct types yourself.

    If a column specification created by [`cols()`](#), it 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, D = date, T = date time, t = time, ? = guess, or _/- to skip the column.

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 use for 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 be trimmed from each field before parsing it?

skip   Number of lines to skip before reading data.

n_max   Maximum number of records to read.

guess_max   Maximum number of records 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 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`.

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("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 [`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("column-types")` 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 supply the correct types yourself.

If a column specification created by [cols()](), it must contain one column speci-
fication for each column. If you only want to read a subset of the columns, use
[cols_only()]().

Unlike other functions type_convert() does not allow character specifications
of col_types.

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

trim_ws   Should leading and trailing whitespace be trimmed from each field before pars-
                  ing 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.

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
type_convert(
  read_csv(readr_example("mtcars.csv"),
           col_types = cols(.default = col_character())))
```

---

write_delim   *Write a data frame to a delimited file*

Description

This is about twice as fast as [write.csv()](), and never writes row names. output_column() is a
generic method used to coerce columns to suitable output.

Usage

```
write_delim(x, path, delim = " ", na = "NA", append = FALSE,
  col_names = !append)

write_csv(x, path, na = "NA", append = FALSE, col_names = !append)
```

```
write_excel_csv(x, path, na = "NA", append = FALSE, col_names = !append)

write_tsv(x, path, na = "NA", append = FALSE, col_names = !append)
```

Arguments

x	A data frame to write to disk
path	Path or connection to write to.
delim	Delimiter used to separate values. Defaults to " ". 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 file does not exist a new file is created.
col_names	Write columns names at the top of the file?

Value

write_*() returns the input x invisibly.

Output

Factors are coerced to character. Doubles are formatted using the grisu3 algorithm. POSIXct's are formatted as ISO8601.

All columns are encoded as UTF-8. `write_excel_csv()` also includes a [UTF-8 Byte order mark](#) which indicates to Excel the csv is UTF-8 encoded.

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

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

```
tmp <- tempfile()
write_csv(mtcars, tmp)
head(read_csv(tmp))

# format_* is useful for testing and reprexes
cat(format_csv(head(mtcars)))
cat(format_tsv(head(mtcars)))
cat(format_delim(head(mtcars), ";"))

df <- data.frame(x = c(1, 2, NA))
format_csv(df, na = ".")

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

```
# A output connection will be automatically created for output filenames
# with appropriate extensions.
dir <- tempdir()
write_tsv(mtcars, file.path(dir, "mtcars.tsv.gz"))
write_tsv(mtcars, file.path(dir, "mtcars.tsv.bz2"))
write_tsv(mtcars, file.path(dir, "mtcars.tsv.xz"))
```

Index