Package ‘meltr’

September 10, 2022

Title Read Non-Rectangular Text Data

Version 1.0.1

Description The goal of 'meltr' is to provide a fast and friendly way to read non-rectangular data, such as ragged forms of csv (comma-separated values), tsv (tab-separated values), and fwf (fixed-width format) files.

License MIT + file LICENSE

URL `https://r-lib.github.io/meltr/`, `https://github.com/r-lib/meltr`

BugReports `https://github.com/r-lib/meltr/issues`

Depends R (>= 2.10)

Imports cli, methods, R6, rlang, tibble

Suggests clipr, covr, crayon, curl, readr, testthat (>= 3.0.0), withr

LinkingTo cpp11

Config/testthat/edition 3

Config/Needs/website dplyr

Encoding UTF-8

RoxygenNote 7.2.1

SystemRequirements C++11

NeedsCompilation yes

Author Hadley Wickham [aut],
Duncan Garmonsway [aut, cre] (@nacnudus),
Jim Hester [aut] (<`https://orcid.org/0000-0002-2739-7082`>),
RStudio [cph, fnd],
https://github.com/mandreyel/ [cph] (mio library)

Maintainer Duncan Garmonsway <`nacnudus@gmail.com`>

Repository CRAN

Date/Publication 2022-09-10 19:52:53 UTC

R topics documented:

`clipboard` *Returns values from the clipboard*

Description

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

Usage

```
clipboard()
```

See Also

readr::read_delim

Examples

```
## Not run:
  clipboard()

## End(Not run)
```

`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 (`https://icu.unicode.org/`) via the stringi package.

Usage

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

date_names_lang(language)

date_names_langs()
```

Arguments

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

Value

A date names object

Examples

```
date_names(mon = LETTERS[1:12], day = letters[1:7])
date_names_lang("en")
date_names_lang("ko")
date_names_lang("fr")
```

`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"
)
```

```
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 - meltr always converts the output to UTF-8.

Value

A locale object

Examples

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

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

`meltr_example` *Get path to meltr example*

Description

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

Usage

```
meltr_example(file = NULL)
```

Arguments

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

Value

A file path or a vector of file names

Examples

```
meltr_example()
meltr_example("mtcars.csv")
```

---

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

Description

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

Usage

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

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

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

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

Arguments

`file`	Either a path to a file, a connection, or literal data (either a single string or a raw vector). 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, the input must be either wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line. Using a value of `clipboard()` will read from the system clipboard.
`delim`	Single character used to separate fields within a record.

`quote`	Single character used to quote strings.
`escape_backslash`	Does the file use backslashes to escape special characters? This is more general than `escape_double` as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like `\\n`.
`escape_double`	Does the file escape quotes by doubling them? i.e. If this option is `TRUE`, the value `""""` represents a single quote, `\"`.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`quoted_na`	**[Deprecated]** Should missing values inside quotes be treated as missing values (the default) or strings. This parameter is soft deprecated as of readr 2.0.0.
`comment`	A string used to identify comments. Any text after the comment characters will be silently ignored.
`trim_ws`	Should leading and trailing whitespace (ASCII spaces and tabs) be trimmed from each field before parsing it?
`skip`	Number of lines to skip before reading data. If `comment` is supplied any commented lines are ignored *after* skipping.
`n_max`	Maximum number of lines to read.
`progress`	Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.
`skip_empty_rows`	Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.

Details

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

Value

A `tibble()` of four columns:

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

- value, the token itself as a character string, unchanged from its representation in the original file.

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

See Also

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

Examples

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

## End(Not run)

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

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

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

---

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

Description

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

Usage

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

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

fwf_widths(widths, col_names = NULL)

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

fwf_cols(...)
```

Arguments

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

`n_max`	Maximum number of lines to read.
`progress`	Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.
`skip_empty_rows`	Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.
`col_names`	Either NULL, or a character vector column names.
`n`	Number of lines the tokenizer will read to determine file structure. By default it is set to 100.
`widths`	Width of each field. Use NA as width of last field when reading a ragged fwf file.
`start, end`	Starting and ending (inclusive) positions of each field. Use NA as last end field when reading a ragged fwf file.
`...`	If the first element is a data frame, then it must have all numeric columns and either one or two rows. The column names are the variable names. The column values are the variable widths if a length one vector, and if length two, variable start and end positions. The elements of `...` are used to construct a data frame with or or two rows as above.

Details

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

Value

A `tibble()` of four columns:

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

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

See Also

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

Examples

```
fwf_sample <- meltr_example("fwf-sample.txt")
writeLines(readLines(fwf_sample))

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

---

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

Description

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

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

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

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

Usage

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

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

Arguments

`file`	Either a path to a file, a connection, or literal data (either a single string or a raw vector). 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, the input must be either wrapped with `I()`, be a string containing at least one new line, or be a vector containing at least one string with a new line. Using a value of `clipboard()` will read from the system clipboard.
`locale`	The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
`na`	Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
`skip`	Number of lines to skip before reading data.
`n_max`	Maximum number of lines to read.
`guess_max`	Maximum number of lines to use for guessing column types. 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.
`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.

Value

A `tibble()` of four columns:

- `row`, the row that the token comes from in the original file
- `col`, the column that the token comes from in the original file

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

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

See Also

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

Examples

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

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

---

problems *Retrieve parsing problems*

Description

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

Usage

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

stop_for_problems(x)
```

Arguments

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

Value

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

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

Examples

```
if (requireNamespace("readr")) {
x <- readr::parse_integer(c("1X", "blah", "3"))
problems(x)

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

`show_progress` *Determine whether progress bars should be shown*

Description

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

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

Usage

```
show_progress()
```

Value

A logical value

Examples

```
show_progress()
```

Index