Package 'charr'

November 7, 2026

Title String Processing Reimagined for 'ALTREP' Strings

Version 0.2.0

Date 2026-11-06

Description An experimental fork of 'stringr'. The functions and semantics are the same, apart from differences between 'ICU' and Unicode versions, but everything is optimized around 'ALTREP' strings (custom high-performance string storage), allowing for faster and more efficient operation.

License MIT + file LICENSE

Copyright Copyright (c) 2026 Travers Ching. Portions derived from the 'stringr' package are copyright (c) 2023 stringr authors; current upstream metadata identifies Hadley Wickham and Posit Software, PBC as copyright holders. Those portions are used under the MIT License. Included R and C++ code from 'stringi' is copyright (c) 2013-2025 Marek Gagolewski and is used under the BSD 3-Clause License. Bundled ICU4C 78.3 source code and data are copyright (c) 2016-2025 Unicode, Inc., with additional holders and terms identified in ICU's third-party notices, under the Unicode License v3 and the applicable component licenses. The source package includes LICENSE.note; installed notices are supplied as COPYRIGHTS and icu/LICENSE.

URL https://github.com/charbase/charr, https://charbase.github.io/charr/

BugReports https://github.com/charbase/charr/issues

Depends R (>= 4.1.0)

Imports charport (>= 0.2.0), cli, glue (>= 1.6.1), lifecycle (>= 1.0.3), rlang (>= 1.0.0), stringi (>= 1.5.3), vctrs (>= 0.4.0)

LinkingTo charport (>= 0.2.0)

Suggests htmltools, htmlwidgets, quarto, testthat (>= 3.0.0)

VignetteBuilder quarto

SystemRequirements GNU make, C++17, ICU4C 78.2 or 78.3 (optional; bundled otherwise)

Config/testthat/edition 3

Config/Needs/website pkgdown, xml2

Encoding UTF-8

LazyData true

Config/roxygen2/version 8.0.0

NeedsCompilation yes

Author Travers Ching [aut, cre, cph],
Hadley Wickham [ctb, cph] (Author of stringr, of which charr is a fork),
Posit Software, PBC [ctb, cph] (Copyright holder of stringr),
Unicode, Inc. and others [ctb, cph] (ICU4C source code and data (Unicode License v3)),
Marek Gagolewski [ctb, cph] (Author of stringi; stringi implementation used by charr),
R Consortium [fnd] (Infrastructure Steering Committee grant: Universal ALTREP Interoperability for Strings)

Maintainer Travers Ching <traversc@gmail.com>

Repository CRAN

Date/Publication 2026-11-07 03:40:02 UTC

Contents

case . . . 3
charr_backend . . . 4
charr_chunking . . . 5
charr_threads . . . 6
invert_match . . . 7
modifiers . . . 7
stringr-data . . . 9
str_c . . . 10
str_conv . . . 11
str_count . . . 12
str_detect . . . 13
str_dup . . . 14
str_equal . . . 14
str_escape . . . 15
str_extract . . . 16
str_flatten . . . 17
str_glue . . . 18
str_length . . . 19
str_like . . . 21
str_locate . . . 22
str_match . . . 23
str_order . . . 24
str_pad . . . 26
str_read_lines . . . 27

str_remove 28
str_replace 29
str_replace_na 30
str_reverse 31
str_split 32
str_starts 34
str_sub 35
str_subset 36
str_to_camel 37
str_trim 38
str_trunc 39
str_unique 40
str_view 41
str_which 42
str_wrap 43
str_write_lines 44
word 45

Index **46**

`case` *Convert string to upper case, lower case, title case, or sentence case*

Description

- `str_to_upper()` converts to upper case.
- `str_to_lower()` converts to lower case.
- `str_to_title()` converts to title case, where only the first letter of each word is capitalized.
- `str_to_sentence()` convert to sentence case, where only the first letter of sentence is capitalized.

Usage

```
str_to_upper(string, locale = "en")

str_to_lower(string, locale = "en")

str_to_title(string, locale = "en")

str_to_sentence(string, locale = "en")
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`locale`	Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.

Value

A character vector the same length as `string`.

Examples

```
dog <- "The quick brown dog"
str_to_upper(dog)
str_to_lower(dog)
str_to_title(dog)
str_to_sentence("the quick brown dog")

# Locale matters!
str_to_upper("i") # English
str_to_upper("i", "tr") # Turkish
```

---

charr_backend *Select charr's string backend*

Description

`charr_backend()` selects the implementation used by charr's public string functions. Dispatch happens once, in R, when a public operation starts. Nested calls stay within the selected backend.

Usage

```
charr_backend(value = NULL)
```

Arguments

`value`	`NULL` to query the current backend, or one of `"reference"`, `"base"`, or `"altrep"` to select a backend.

Details

The available backends are:

- `"reference"`: the original stringr implementation, which calls stringi.
- `"base"`: charr's optimized backend returning ordinary R character vectors.
- `"altrep"`: charr's optimized charport backend returning ALTREP character vectors where appropriate. This is the default.

The selection is stored in the global `charr_backend` option. Changing the option affects later public calls; it does not change an operation already in progress.

Value

The current backend when querying. When setting, the previous backend is returned invisibly.

Examples

```
old <- charr_backend("base")
charr_backend()
charr_backend(old)
```

`charr_chunking` *Tune how parallel work is divided*

Description

These control how a data-parallel operation cuts its work into chunks once `charr_threads()` has decided how many threads to use. Neither affects whether threads are used at all, and neither changes any result.

Usage

```
charr_chunks_per_worker(value = NULL)

charr_min_chunk(value = NULL)
```

Arguments

`value`	`NULL` to query the current setting, or a positive whole number.

Details

Threads draw chunks one at a time rather than taking a fixed slice each, so a thread that finishes a cheap chunk comes back for another. That is what keeps uneven input balanced: element cost is not uniform, and one long string can outweigh a thousand short ones.

`charr_chunks_per_worker()` sets how many chunks each thread should get, defaulting to `128`. Higher values balance skewed input better and cost a little more bookkeeping. `charr_min_chunk()` sets the smallest number of elements a chunk may hold, defaulting to `256`, which stops a large thread count from cutting the work finer than it is worth handing out.

With `n` elements and `w` threads the chunk size is `ceiling(n / (w * chunks_per_worker))`, raised to at least `min_chunk` and then lowered if needed so that there are never fewer chunks than threads. A short vector therefore still spreads across every thread.

Like `charr_threads()`, both settings are held in native code rather than in an R option, because they are only ever read there. These accessors are the only way to change them.

Value

The current setting when querying. When setting, the previous value is returned invisibly.

Examples

```
old <- charr_chunks_per_worker(256)
charr_chunks_per_worker()
charr_chunks_per_worker(old)
```

`charr_threads` *Set the number of worker threads*

Description

`charr_threads()` sets how many threads charr's `"altrep"` backend may use for the operations that have a data-parallel shape. Operations without that shape, and the `"base"` and `"reference"` backends, always run on one thread.

Usage

```
charr_threads(value = NULL)
```

Arguments

`value`	`NULL` to query the current count, or a positive whole number of threads. `parallel::detectCores()` is the usual source of an upper bound.

Details

An eligible operation uses the requested count, capped by its number of tasks and an internal safety limit of 256. Charr does not apply an input-size heuristic, so even a small eligible input starts workers when the count is greater than one. A few modes must stay serial because their results depend on the order in which elements run.

The count defaults to 1, so charr is single-threaded until asked otherwise. It is held in native code rather than in an R option, because threading happens only in native code. This accessor is the only way to change it: setting an option named `charr_threads` has no effect.

Value

The current count when querying. When setting, the previous count is returned invisibly.

Examples

```
old <- charr_threads(2)
charr_threads()
charr_threads(old)
```

invert_match *Switch location of matches to location of non-matches*

Description

Invert a matrix of match locations to match the opposite of what was previously matched.

Usage

```
invert_match(loc)
```

Arguments

loc	matrix of match locations, as from str_locate_all()

Value

numeric match giving locations of non-matches

Examples

```
numbers <- "1 and 2 and 4 and 456"
num_loc <- str_locate_all(numbers, "[0-9]+")[[1]]
str_sub(numbers, num_loc[, "start"], num_loc[, "end"])

text_loc <- invert_match(num_loc)
str_sub(numbers, text_loc[, "start"], text_loc[, "end"])
```

modifiers *Control matching behaviour with modifier functions*

Description

Modifier functions control the meaning of the `pattern` argument to stringr functions:

- `boundary()`: Match boundaries between things.
- `coll()`: Compare strings using standard Unicode collation rules.
- `fixed()`: Compare literal bytes.
- `regex()` (the default): Uses ICU regular expressions.

Usage

```
fixed(pattern, ignore_case = FALSE)

coll(pattern, ignore_case = FALSE, locale = "en", ...)

regex(
  pattern,
  ignore_case = FALSE,
  multiline = FALSE,
  comments = FALSE,
  dotall = FALSE,
  ...
)

boundary(
  type = c("character", "line_break", "sentence", "word"),
  skip_word_none = NA,
  ...
)
```

Arguments

pattern	Pattern to modify behaviour.
ignore_case	Should case differences be ignored in the match? For `fixed()`, this uses a simple algorithm which assumes a one-to-one mapping between upper and lower case letters.
locale	Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.
...	Other less frequently used arguments passed on to `stringi::stri_opts_collator()`, `stringi::stri_opts_regex()`, or `stringi::stri_opts_brkiter()`
multiline	If `TRUE`, `$` and `^` match the beginning and end of each line. If `FALSE`, the default, only match the start and end of the input.
comments	If `TRUE`, white space and comments beginning with # are ignored. Escape literal spaces with `\\ `.
dotall	If `TRUE`, `.` will also match line terminators.
type	Boundary type to detect. `character` Every character is a boundary. `line_break` Boundaries are places where it is acceptable to have a line break in the current locale. `sentence` The beginnings and ends of sentences are boundaries, using intelligent rules to avoid counting abbreviations (details). `word` The beginnings and ends of words are boundaries.
skip_word_none	Ignore "words" that don't contain any characters or numbers - i.e. punctuation. Default `NA` will skip such "words" only when splitting on `word` boundaries.

Value

A stringr modifier object, i.e. a character vector with parent S3 class `stringr_pattern`.

Examples

```
pattern <- "a.b"
strings <- c("abb", "a.b")
str_detect(strings, pattern)
str_detect(strings, fixed(pattern))
str_detect(strings, coll(pattern))

# coll() is useful for locale-aware case-insensitive matching
i <- c("I", "\u0130", "i")
i
str_detect(i, fixed("i", TRUE))
str_detect(i, coll("i", TRUE))
str_detect(i, coll("i", TRUE, locale = "tr"))

# Word boundaries
words <- c("These are   some words.")
str_count(words, boundary("word"))
str_split(words, " ")[[1]]
str_split(words, boundary("word"))[[1]]

# Regular expression variations
str_extract_all("The Cat in the Hat", "[a-z]+")
str_extract_all("The Cat in the Hat", regex("[a-z]+", TRUE))

str_extract_all("a\nb\nc", "^.")
str_extract_all("a\nb\nc", regex("^.", multiline = TRUE))

str_extract_all("a\nb\nc", "a.")
str_extract_all("a\nb\nc", regex("a.", dotall = TRUE))
```

`stringr-data` *Sample character vectors for practicing string manipulations*

Description

`fruit` and `words` come from the `rcorpora` package written by Gabor Csardi; the data was collected by Darius Kazemi and made available at https://github.com/dariusk/corpora. `sentences` is a collection of "Harvard sentences" used for standardised testing of voice.

Usage

```
sentences

fruit

words
```

Format

Character vectors.

Examples

```
length(sentences)
sentences[1:5]

length(fruit)
fruit[1:5]

length(words)
words[1:5]
```

---

str_c *Join multiple strings into one string*

Description

`str_c()` combines multiple character vectors into a single character vector. It's very similar to `paste0()` but uses tidyverse recycling and `NA` rules.

One way to understand how `str_c()` works is picture a 2d matrix of strings, where each argument forms a column. `sep` is inserted between each column, and then each row is combined together into a single string. If `collapse` is set, it's inserted between each row, and then the result is again combined, this time into a single string.

Usage

```
str_c(..., sep = "", collapse = NULL)
```

Arguments

`...`	One or more character vectors. `NULL`s are removed; scalar inputs (vectors of length 1) are recycled to the common length of vector inputs. Like most other R functions, missing values are "infectious": whenever a missing value is combined with another string the result will always be missing. Use `str_replace_na()` or `dplyr::coalesce()` to convert to the desired value.
`sep`	String to insert between input vectors.
`collapse`	Optional string used to combine output into single string. Generally better to use `str_flatten()` if you needed this behaviour.

Value

If `collapse = NULL` (the default) a character vector with length equal to the longest input. If `collapse` is a string, a character vector of length 1.

Examples

```
str_c("Letter: ", letters)
str_c("Letter", letters, sep = ": ")
str_c(letters, " is for", "...")
str_c(letters[-26], " comes before ", letters[-1])

str_c(letters, collapse = "")
str_c(letters, collapse = ", ")

# Differences from paste() ----------------------
# Missing inputs give missing outputs
str_c(c("a", NA, "b"), "-d")
paste0(c("a", NA, "b"), "-d")
# Use str_replace_NA to display literal NAs:
str_c(str_replace_na(c("a", NA, "b")), "-d")

# Uses tidyverse recycling rules
## Not run: str_c(1:2, 1:3) # errors
paste0(1:2, 1:3)

str_c("x", character())
paste0("x", character())
```

---

str_conv *Specify the encoding of a string*

Description

This is a convenient way to override the current encoding of a string.

Usage

```
str_conv(string, encoding)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`encoding`	Name of encoding. See `stringi::stri_enc_list()` for a complete list.

Value

A character vector the same length as `string`, converted from `encoding` to UTF-8.

Examples

```
# Example from encoding?stringi::stringi
x <- rawToChar(as.raw(177))
x
str_conv(x, "ISO-8859-2") # Polish "a with ogonek"
str_conv(x, "ISO-8859-1") # Plus-minus
```

`str_count` *Count number of matches*

Description

Counts the number of times `pattern` is found within each element of `string`.

Usage

```
str_count(string, pattern = "")
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. Match character, word, line and sentence boundaries with `boundary()`. The empty string, `""`, is equivalent to `boundary("character")`.

Value

An integer vector the same length as `string`/`pattern`.

See Also

`stringi::stri_count()` which this function wraps.

`str_locate()`/`str_locate_all()` to locate position of matches

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_count(fruit, "a")
str_count(fruit, "p")
str_count(fruit, "e")
str_count(fruit, c("a", "b", "p", "p"))

str_count(c("a.", "...", ".a.a"), ".")
str_count(c("a.", "...", ".a.a"), fixed("."))
```

`str_detect` *Detect the presence/absence of a match*

Description

`str_detect()` returns a logical vector with `TRUE` for each element of `string` that matches `pattern` and `FALSE` otherwise. It's equivalent to `grepl(pattern, string)`.

Usage

```
str_detect(string, pattern, negate = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. You can not match boundaries, including `""`, with this function.
`negate`	If `TRUE`, inverts the resulting boolean vector.

Value

A logical vector the same length as `string`/`pattern`.

See Also

`stringi::stri_detect()` which this function wraps, `str_subset()` for a convenient wrapper around `x[str_detect(x, pattern)]`

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_detect(fruit, "a")
str_detect(fruit, "^a")
str_detect(fruit, "a$")
str_detect(fruit, "b")
str_detect(fruit, "[aeiou]")

# Also vectorised over pattern
str_detect("aecfg", letters)

# Returns TRUE if the pattern do NOT match
str_detect(fruit, "^p", negate = TRUE)
```

str_dup *Duplicate a string*

Description

`str_dup()` duplicates the characters within a string, e.g. `str_dup("xy", 3)` returns `"xyxyxy"`.

Usage

```
str_dup(string, times, sep = NULL)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`times`	Number of times to duplicate each string.
`sep`	String to insert between each duplicate.

Value

A character vector the same length as `string`/`times`.

Examples

```
fruit <- c("apple", "pear", "banana")
str_dup(fruit, 2)
str_dup(fruit, 2, sep = " ")
str_dup(fruit, 1:3)
str_c("ba", str_dup("na", 0:5))
```

str_equal *Determine if two strings are equivalent*

Description

This uses Unicode canonicalisation rules, and optionally ignores case.

Usage

```
str_equal(x, y, locale = "en", ignore_case = FALSE, ...)
```

Arguments

`x, y`	A pair of character vectors.
`locale`	Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.
`ignore_case`	Ignore case when comparing strings?
`...`	Other options used to control collation. Passed on to `stringi::stri_opts_collator()`.

Value

An logical vector the same length as x/y.

See Also

`stringi::stri_cmp_equiv()` for the underlying implementation.

Examples

```
# These two strings encode "a" with an accent in two different ways
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)

a1 == a2
str_equal(a1, a2)

# ohm and omega use different code points but should always be treated
# as equal
ohm <- "\u2126"
omega <- "\u03A9"
c(ohm, omega)

ohm == omega
str_equal(ohm, omega)
```

---

str_escape *Escape regular expression metacharacters*

Description

This function escapes metacharacter, the characters that have special meaning to the regular expression engine. In most cases you are better off using `fixed()` since it is faster, but `str_escape()` is useful if you are composing user provided strings into a pattern.

Usage

```
str_escape(string)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.

Value

A character vector the same length as `string`.

Examples

```
str_detect(c("a", "."), ".")
str_detect(c("a", "."), str_escape("."))
```

`str_extract` *Extract the complete match*

Description

`str_extract()` extracts the first complete match from each string, `str_extract_all()`extracts all matches from each string.

Usage

```
str_extract(string, pattern, group = NULL)

str_extract_all(string, pattern, simplify = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. Match character, word, line and sentence boundaries with `boundary()`. The empty string, `""`, is equivalent to `boundary("character")`.
`group`	If supplied, instead of returning the complete match, will return the matched text from the specified capturing group.
`simplify`	A boolean. • `FALSE` (the default): returns a list of character vectors. • `TRUE`: returns a character matrix.

Value

- `str_extract()`: a character vector the same length as `string`/`pattern`.
- `str_extract_all()`: a list of character vectors the same length as `string`/`pattern`.

See Also

`str_match()` to extract matched groups; `stringi::stri_extract()` for the underlying implementation.

Examples

```
shopping_list <- c("apples x4", "bag of flour", "bag of sugar", "milk x2")
str_extract(shopping_list, "\\d")
str_extract(shopping_list, "[a-z]+")
str_extract(shopping_list, "[a-z]{1,4}")
str_extract(shopping_list, "\\b[a-z]{1,4}\\b")

str_extract(shopping_list, "([a-z]+) of ([a-z]+)")
str_extract(shopping_list, "([a-z]+) of ([a-z]+)", group = 1)
str_extract(shopping_list, "([a-z]+) of ([a-z]+)", group = 2)

# Extract all matches
str_extract_all(shopping_list, "[a-z]+")
str_extract_all(shopping_list, "\\b[a-z]+\\b")
str_extract_all(shopping_list, "\\d")

# Simplify results into character matrix
str_extract_all(shopping_list, "\\b[a-z]+\\b", simplify = TRUE)
str_extract_all(shopping_list, "\\d", simplify = TRUE)

# Extract all words
str_extract_all("This is, suprisingly, a sentence.", boundary("word"))
```

---

str_flatten *Flatten a string*

Description

`str_flatten()` reduces a character vector to a single string. This is a summary function because regardless of the length of the input `x`, it always returns a single string.

`str_flatten_comma()` is a variation designed specifically for flattening with commas. It automatically recognises if `last` uses the Oxford comma and handles the special case of 2 elements.

Usage

```
str_flatten(string, collapse = "", last = NULL, na.rm = FALSE)

str_flatten_comma(string, last = NULL, na.rm = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`collapse`	String to insert between each piece. Defaults to `""`.
`last`	Optional string to use in place of the final separator.
`na.rm`	Remove missing values? If `FALSE` (the default), the result will be `NA` if any element of `string` is `NA`.

Value

A string, i.e. a character vector of length 1.

Examples

```
str_flatten(letters)
str_flatten(letters, "-")

str_flatten(letters[1:3], ", ")

# Use last to customise the last component
str_flatten(letters[1:3], ", ", " and ")

# this almost works if you want an Oxford (aka serial) comma
str_flatten(letters[1:3], ", ", ", and ")

# but it will always add a comma, even when not necessary
str_flatten(letters[1:2], ", ", ", and ")

# str_flatten_comma knows how to handle the Oxford comma
str_flatten_comma(letters[1:3], ", and ")
str_flatten_comma(letters[1:2], ", and ")
```

---

str_glue *Interpolation with glue*

Description

These functions are wrappers around `glue::glue()` and `glue::glue_data()`, which provide a powerful and elegant syntax for interpolating strings with `{}`.

These wrappers provide a small set of the full options. Use `glue()` and `glue_data()` directly from glue for more control.

Usage

```
str_glue(..., .sep = "", .envir = parent.frame(), .trim = TRUE)

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA")
```

Arguments

`...`	[expressions] Unnamed arguments are taken to be expression string(s) to format. Multiple inputs are concatenated together before formatting. Named arguments are taken to be temporary variables available for substitution. For `glue_data()`, elements in `...` override the values in `.x`.
`.sep`	[`character(1)`: `""`] Separator used to separate elements.

`.envir` [environment: parent.frame()]
Environment to evaluate each expression in. Expressions are evaluated from left to right. If `.x` is an environment, the expressions are evaluated in that environment and `.envir` is ignored. If NULL is passed, it is equivalent to `emptyenv()`.

`.trim` [logical(1): 'TRUE']
Whether to trim the input template with `trim()` or not.

`.x` [listish]
An environment, list, or data frame used to lookup values.

`.na` [character(1): 'NA']
Value to replace NA values with. If NULL missing values are propagated, that is an NA result will cause NA output. Otherwise the value is replaced by the value of `.na`.

Value

A character vector with same length as the longest input.

Examples

```
name <- "Fred"
age <- 50
anniversary <- as.Date("1991-10-12")
str_glue(
  "My name is {name}, ",
  "my age next year is {age + 1}, ",
  "and my anniversary is {format(anniversary, '%A, %B %d, %Y')}."
)

# single braces can be inserted by doubling them
str_glue("My name is {name}, not {{name}}.")

# You can also used named arguments
str_glue(
  "My name is {name}, ",
  "and my age next year is {age + 1}.",
  name = "Joe",
  age = 40
)

# `str_glue_data()` is useful in data pipelines
mtcars |> str_glue_data("{rownames(mtcars)} has {hp} hp")
```

`str_length` *Compute the length/width*

Description

`str_length()` returns the number of codepoints in a string. These are the individual elements (which are often, but not always letters) that can be extracted with `str_sub()`.

`str_width()` returns how much space the string will occupy when printed in a fixed width font (i.e. when printed in the console).

Usage

```
str_length(string)

str_width(string)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.

Value

A numeric vector the same length as `string`.

See Also

`stringi::stri_length()` which this function wraps.

Examples

```
str_length(letters)
str_length(NA)
str_length(factor("abc"))
str_length(c("i", "like", "programming", NA))

# Some characters, like emoji and Chinese characters (hanzi), are square
# which means they take up the width of two Latin characters
x <- c("\u6c49\u5b57", "\U0001f60a")
str_view(x)
str_width(x)
str_length(x)

# There are two ways of representing a u with an umlaut
u <- c("\u00fc", "u\u0308")
# They have the same width
str_width(u)
# But a different length
str_length(u)
# Because the second element is made up of a u + an accent
str_sub(u, 1, 1)
```

str_like *Detect a pattern in the same way as* SQL*'s* LIKE *and* ILIKE *operators*

Description

`str_like()` and `str_like()` follow the conventions of the SQL `LIKE` and `ILIKE` operators, namely:

- Must match the entire string.
- _ matches a single character (like `.`).
- % matches any number of characters (like `.*`).
- \% and _ match literal % and _.

The difference between the two functions is their case-sensitivity: `str_like()` is case sensitive and `str_ilike()` is not.

Usage

```
str_like(string, pattern, ignore_case = deprecated())

str_ilike(string, pattern)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	A character vector containing a SQL "like" pattern. See above for details.
`ignore_case`	**[Deprecated]**

Value

A logical vector the same length as `string`.

Note

Prior to stringr 1.6.0, `str_like()` was incorrectly case-insensitive.

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_like(fruit, "app")
str_like(fruit, "app%")
str_like(fruit, "APP%")
str_like(fruit, "ba_ana")
str_like(fruit, "%apple")

str_ilike(fruit, "app")
str_ilike(fruit, "app%")
str_ilike(fruit, "APP%")
str_ilike(fruit, "ba_ana")
str_ilike(fruit, "%apple")
```

`str_locate` *Find location of match*

Description

`str_locate()` returns the `start` and `end` position of the first match; `str_locate_all()` returns the `start` and `end` position of each match.

Because the `start` and `end` values are inclusive, zero-length matches (e.g. `$`, `^`, `\\b`) will have an `end` that is smaller than `start`.

Usage

```
str_locate(string, pattern)

str_locate_all(string, pattern)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. Match character, word, line and sentence boundaries with `boundary()`. The empty string, `""`, is equivalent to `boundary("character")`.

Value

- `str_locate()` returns an integer matrix with two columns and one row for each element of `string`. The first column, `start`, gives the position at the start of the match, and the second column, `end`, gives the position of the end.
- `str_locate_all()` returns a list of integer matrices with the same length as `string`/`pattern`. The matrices have columns `start` and `end` as above, and one row for each match.

See Also

`str_extract()` for a convenient way of extracting matches, `stringi::stri_locate()` for the underlying implementation.

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_locate(fruit, "$")
str_locate(fruit, "a")
str_locate(fruit, "e")
str_locate(fruit, c("a", "b", "p", "p"))

str_locate_all(fruit, "a")
str_locate_all(fruit, "e")
str_locate_all(fruit, c("a", "b", "p", "p"))

# Find location of every character
str_locate_all(fruit, "")
```

---

str_match *Extract components (capturing groups) from a match*

Description

Extract any number of matches defined by unnamed, `(pattern)`, and named, `(?<name>pattern)` capture groups.

Use a non-capturing group, `(?:pattern)`, if you need to override default operate precedence but don't want to capture the result.

Usage

```
str_match(string, pattern)

str_match_all(string, pattern)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Unlike other stringr functions, `str_match()` only supports regular expressions, as described in stringr's regular expressions article. The pattern should contain at least one capturing group.

Value

- `str_match()`: a character matrix with the same number of rows as the length of `string`/`pattern`. The first column is the complete match, followed by one column for each capture group. The columns will be named if you used "named captured groups", i.e. `(?<name>pattern')`.
- `str_match_all()`: a list of the same length as `string`/`pattern` containing character matrices. Each matrix has columns as described above and one row for each match.

See Also

str_extract() to extract the complete match, stringi::stri_match() for the underlying implementation.

Examples

```
strings <- c(" 219 733 8965", "329-293-8753 ", "banana", "595 794 7569",
  "387 287 6718", "apple", "233.398.9187  ", "482 952 3315",
  "239 923 8115 and 842 566 4692", "Work: 579-499-7527", "$1000",
  "Home: 543.355.3679")
phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"

str_extract(strings, phone)
str_match(strings, phone)

# Extract/match all
str_extract_all(strings, phone)
str_match_all(strings, phone)

# You can also name the groups to make further manipulation easier
phone <- "(?<area>[2-9][0-9]{2})[- .](?<phone>[0-9]{3}[- .][0-9]{4})"
str_match(strings, phone)

x <- c("<a> <b>", "<a> <>", "<a>", "", NA)
str_match(x, "<(.*?)> <(.*?)>")
str_match_all(x, "<(.*?)>")

str_extract(x, "<.*?>")
str_extract_all(x, "<.*?>")
```

---

str_order *Order, rank, or sort a character vector*

Description

- str_sort() returns the sorted vector.
- str_order() returns an integer vector that returns the desired order when used for subsetting, i.e. x[str_order(x)] is the same as str_sort()
- str_rank() returns the ranks of the values, i.e. arrange(df, str_rank(x)) is the same as str_sort(df$x).

Usage

```
str_order(
  x,
  decreasing = FALSE,
  na_last = TRUE,
  locale = "en",
```

```
  numeric = FALSE,
  ...
)

str_rank(x, locale = "en", numeric = FALSE, ...)

str_sort(
  x,
  decreasing = FALSE,
  na_last = TRUE,
  locale = "en",
  numeric = FALSE,
  ...
)
```

Arguments

x	A character vector to sort.
decreasing	A boolean. If FALSE, the default, sorts from lowest to highest; if TRUE sorts from highest to lowest.
na_last	Where should NA go? TRUE at the end, FALSE at the beginning, NA dropped.
locale	Locale to use for comparisons. See stringi::stri_locale_list() for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.
numeric	If TRUE, will sort digits numerically, instead of as strings.
...	Other options used to control collation. Passed on to stringi::stri_opts_collator().

Value

A character vector the same length as string.

See Also

stringi::stri_order() for the underlying implementation.

Examples

```
x <- c("apple", "car", "happy", "char")
str_sort(x)

str_order(x)
x[str_order(x)]

str_rank(x)

# In Czech, ch is a digraph that sorts after h
str_sort(x, locale = "cs")

# Use numeric = TRUE to sort numbers in strings
```

```
x <- c("100a10", "100a5", "2b", "2a")
str_sort(x)
str_sort(x, numeric = TRUE)
```

`str_pad` *Pad a string to minimum width*

Description

Pad a string to a fixed width, so that `str_length(str_pad(x, n))` is always greater than or equal to `n`.

Usage

```
str_pad(
  string,
  width,
  side = c("left", "right", "both"),
  pad = " ",
  use_width = TRUE
)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`width`	Minimum width of padded strings.
`side`	Side on which padding character is added (left, right or both).
`pad`	Single padding character (default is a space).
`use_width`	If `FALSE`, use the length of the string instead of the width; see `str_width()`/`str_length()` for the difference.

Value

A character vector the same length as `string`/`width`/`pad`.

See Also

`str_trim()` to remove whitespace; `str_trunc()` to decrease the maximum width of a string.

Examples

```
rbind(
  str_pad("hadley", 30, "left"),
  str_pad("hadley", 30, "right"),
  str_pad("hadley", 30, "both")
)
```

```
# All arguments are vectorised except side
str_pad(c("a", "abc", "abcdef"), 10)
str_pad("a", c(5, 10, 20))
str_pad("a", 10, pad = c("-", "_", " "))

# Longer strings are returned unchanged
str_pad("hadley", 3)
```

`str_read_lines` *Read a text file as Unicode lines*

Description

`str_read_lines()` reads a text file in its entirety, converts it to UTF-8, and splits it at Unicode line boundaries.

Usage

```
str_read_lines(con, encoding = NULL)
```

Arguments

`con`	A file name or a connection opened in binary mode.
`encoding`	A single string giving the input encoding. `NULL` or `""` uses the current default encoding.

Value

A character vector with one element per line.

See Also

`stringi::stri_read_lines()` for the underlying implementation.

Examples

```
path <- tempfile()
writeLines(c("first", "second"), path, useBytes = TRUE)
str_read_lines(path, encoding = "UTF-8")
unlink(path)
```

`str_remove` *Remove matched patterns*

Description

Remove matches, i.e. replace them with `""`.

Usage

```
str_remove(string, pattern)

str_remove_all(string, pattern)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. You can not match boundaries, including `""`, with this function.

Value

A character vector the same length as `string`/`pattern`.

See Also

`str_replace()` for the underlying implementation.

Examples

```
fruits <- c("one apple", "two pears", "three bananas")
str_remove(fruits, "[aeiou]")
str_remove_all(fruits, "[aeiou]")
```

`str_replace` *Replace matches with new text*

Description

`str_replace()` replaces the first match; `str_replace_all()` replaces all matches.

Usage

```
str_replace(string, pattern, replacement)

str_replace_all(string, pattern, replacement)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringi::about_search_regex. Control options with `regex()`. For `str_replace_all()` this can also be a named vector (`c(pattern1 = replacement1)`), in order to perform multiple replacements in each element of `string`. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. You can not match boundaries, including `""`, with this function.
`replacement`	The replacement value, usually a single string, but it can be the a vector the same length as `string` or `pattern`. References of the form `\1`, `\2`, etc will be replaced with the contents of the respective matched group (created by `()`). Alternatively, supply a function (or formula): it will be passed a single character vector and should return a character vector of the same length. To replace the complete string with `NA`, use `replacement = NA_character_`.

Value

A character vector the same length as `string`/`pattern`/`replacement`.

See Also

`str_replace_na()` to turn missing values into "NA"; `stringi::stri_replace()` for the underlying implementation.

Examples

```
fruits <- c("one apple", "two pears", "three bananas")
str_replace(fruits, "[aeiou]", "-")
str_replace_all(fruits, "[aeiou]", "-")
str_replace_all(fruits, "[aeiou]", toupper)
str_replace_all(fruits, "b", NA_character_)

str_replace(fruits, "([aeiou])", "")
str_replace(fruits, "([aeiou])", "\\1\\1")

# Note that str_replace() is vectorised along text, pattern, and replacement
str_replace(fruits, "[aeiou]", c("1", "2", "3"))
str_replace(fruits, c("a", "e", "i"), "-")

# If you want to apply multiple patterns and replacements to the same
# string, pass a named vector to pattern.
fruits |>
  str_c(collapse = "---") |>
  str_replace_all(c("one" = "1", "two" = "2", "three" = "3"))

# Use a function for more sophisticated replacement. This example
# replaces colour names with their hex values.
colours <- str_c("\\b", colors(), "\\b", collapse="|")
col2hex <- function(col) {
  rgb <- col2rgb(col)
  rgb(rgb["red", ], rgb["green", ], rgb["blue", ], maxColorValue = 255)
}

x <- c(
  "Roses are red, violets are blue",
  "My favourite colour is green"
)
str_replace_all(x, colours, col2hex)
```

---

str_replace_na *Turn NA into "NA"*

Description

Turn NA into "NA"

Usage

```
str_replace_na(string, replacement = "NA")
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
replacement	A single string.

Value

A character vector the same length as `string`, with missing values replaced by `replacement`.

Examples

```
str_replace_na(c(NA, "abc", "def"))
```

`str_reverse` *Reverse strings by code point*

Description

`str_reverse()` reverses the order of the Unicode code points in each string.

Usage

```
str_reverse(string)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.

Value

A character vector the same length as `string`.

See Also

`stringi::stri_reverse()` for the underlying implementation.

Examples

```
str_reverse(c("abc", "caf\u00e9", NA))
```

`str_split` *Split up a string into pieces*

Description

This family of functions provides various ways of splitting a string up into pieces. These two functions return a character vector:

- `str_split_1()` takes a single string and splits it into pieces, returning a single character vector.
- `str_split_i()` splits each string in a character vector into pieces and extracts the `i`th value, returning a character vector.

These two functions return a more complex object:

- `str_split()` splits each string in a character vector into a varying number of pieces, returning a list of character vectors.
- `str_split_fixed()` splits each string in a character vector into a fixed number of pieces, returning a character matrix.

Usage

```
str_split(string, pattern, n = Inf, simplify = FALSE)

str_split_1(string, pattern)

str_split_fixed(string, pattern, n)

str_split_i(string, pattern, i)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. Match character, word, line and sentence boundaries with `boundary()`. The empty string, `""`, is equivalent to `boundary("character")`.
`n`	Maximum number of pieces to return. Default (Inf) uses all possible split positions. For `str_split()`, this determines the maximum length of each element of the output. For `str_split_fixed()`, this determines the number of columns in the output; if an input is too short, the result will be padded with `""`.

simplify A boolean.

- FALSE (the default): returns a list of character vectors.
- TRUE: returns a character matrix.

i Element to return. Use a negative value to count from the right hand side.

Value

- str_split_1(): a character vector.
- str_split(): a list the same length as string/pattern containing character vectors.
- str_split_fixed(): a character matrix with n columns and the same number of rows as the length of string/pattern.
- str_split_i(): a character vector the same length as string/pattern.

See Also

stringi::stri_split() for the underlying implementation.

Examples

```
fruits <- c(
  "apples and oranges and pears and bananas",
  "pineapples and mangos and guavas"
)

str_split(fruits, " and ")
str_split(fruits, " and ", simplify = TRUE)

# If you want to split a single string, use `str_split_1`
str_split_1(fruits[[1]], " and ")

# Specify n to restrict the number of possible matches
str_split(fruits, " and ", n = 3)
str_split(fruits, " and ", n = 2)
# If n greater than number of pieces, no padding occurs
str_split(fruits, " and ", n = 5)

# Use fixed to return a character matrix
str_split_fixed(fruits, " and ", 3)
str_split_fixed(fruits, " and ", 4)

# str_split_i extracts only a single piece from a string
str_split_i(fruits, " and ", 1)
str_split_i(fruits, " and ", 4)
# use a negative number to select from the end
str_split_i(fruits, " and ", -1)
```

str_starts *Detect the presence/absence of a match at the start/end*

Description

`str_starts()` and `str_ends()` are special cases of `str_detect()` that only match at the beginning or end of a string, respectively.

Usage

```
str_starts(string, pattern, negate = FALSE)

str_ends(string, pattern, negate = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern with which the string starts or ends. The default interpretation is a regular expression, as described in stringi::about_search_regex. Control options with `regex()`. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale.
`negate`	If `TRUE`, inverts the resulting boolean vector.

Value

A logical vector.

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_starts(fruit, "p")
str_starts(fruit, "p", negate = TRUE)
str_ends(fruit, "e")
str_ends(fruit, "e", negate = TRUE)
```

str_sub *Get and set substrings using their positions*

Description

`str_sub()` extracts or replaces the elements at a single position in each string. `str_sub_all()` allows you to extract strings at multiple elements in every string.

Usage

```
str_sub(string, start = 1L, end = -1L)

str_sub(string, start = 1L, end = -1L, omit_na = FALSE) <- value

str_sub_all(string, start = 1L, end = -1L)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`start, end`	A pair of integer vectors defining the range of characters to extract (inclusive). Positive values count from the left of the string, and negative values count from the right. In other words, if `string` is `"abcdef"` then 1 refers to `"a"` and -1 refers to `"f"`. Alternatively, instead of a pair of vectors, you can pass a matrix to `start`. The matrix should have two columns, either labelled `start` and `end`, or `start` and `length`. This makes `str_sub()` work directly with the output from `str_locate()` and friends.
`omit_na`	Single logical value. If `TRUE`, missing values in any of the arguments provided will result in an unchanged input.
`value`	Replacement string.

Value

- `str_sub()`: A character vector the same length as `string`/`start`/`end`.
- `str_sub_all()`: A list the same length as `string`. Each element is a character vector the same length as `start`/`end`.

If `end` comes before `start` or `start` is outside the range of `string` then the corresponding output will be the empty string.

See Also

The underlying implementation in `stringi::stri_sub()`

Examples

```
hw <- "Hadley Wickham"

str_sub(hw, 1, 6)
str_sub(hw, end = 6)
str_sub(hw, 8, 14)
str_sub(hw, 8)

# Negative values index from end of string
str_sub(hw, -1)
str_sub(hw, -7)
str_sub(hw, end = -7)

# str_sub() is vectorised by both string and position
str_sub(hw, c(1, 8), c(6, 14))

# if you want to extract multiple positions from multiple strings,
# use str_sub_all()
x <- c("abcde", "ghifgh")
str_sub(x, c(1, 2), c(2, 4))
str_sub_all(x, start = c(1, 2), end = c(2, 4))

# Alternatively, you can pass in a two column matrix, as in the
# output from str_locate_all
pos <- str_locate_all(hw, "[aeio]")[[1]]
pos
str_sub(hw, pos)

# You can also use `str_sub()` to modify strings:
x <- "BBCDEF"
str_sub(x, 1, 1) <- "A"; x
str_sub(x, -1, -1) <- "K"; x
str_sub(x, -2, -2) <- "GHIJ"; x
str_sub(x, 2, -2) <- ""; x
```

---

str_subset *Find matching elements*

Description

`str_subset()` returns all elements of `string` where there's at least one match to `pattern`. It's a wrapper around `x[str_detect(x, pattern)]`, and is equivalent to `grep(pattern, x, value = TRUE)`.

Use `str_extract()` to find the location of the match *within* each string.

Usage

```
str_subset(string, pattern, negate = FALSE)
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
pattern	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use regex() for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using fixed(). This is fast, but approximate. Generally, for matching human text, you'll want coll() which respects character matching rules for the specified locale. You can not match boundaries, including "", with this function.
negate	If TRUE, inverts the resulting boolean vector.

Value

A character vector, usually smaller than string.

See Also

grep() with argument value = TRUE, stringi::stri_subset() for the underlying implementation.

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_subset(fruit, "a")

str_subset(fruit, "^a")
str_subset(fruit, "a$")
str_subset(fruit, "b")
str_subset(fruit, "[aeiou]")

# Elements that don't match
str_subset(fruit, "^p", negate = TRUE)

# Missings never match
str_subset(c("a", NA, "b"), ".")
```

---

str_to_camel *Convert between different types of programming case*

Description

- str_to_camel() converts to camel case, where the first letter of each word is capitalized, with no separation between words. By default the first letter of the first word is not capitalized.
- str_to_kebab() converts to kebab case, where words are converted to lower case and separated by dashes (-).
- str_to_snake() converts to snake case, where words are converted to lower case and separated by underscores (_).

Usage

```
str_to_camel(string, first_upper = FALSE)

str_to_snake(string)

str_to_kebab(string)
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
first_upper	Logical. Should the first letter be capitalized?

Value

A character vector the same length as string.

Examples

```
str_to_camel("my-variable")
str_to_camel("my-variable", first_upper = TRUE)

str_to_snake("MyVariable")
str_to_kebab("MyVariable")
```

---

str_trim *Remove whitespace*

Description

str_trim() removes whitespace from start and end of string; str_squish() removes whitespace at the start and end, and replaces all internal whitespace with a single space.

Usage

```
str_trim(string, side = c("both", "left", "right"))

str_squish(string)
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
side	Side on which to remove whitespace: "left", "right", or "both", the default.

Value

A character vector the same length as string.

See Also

str_pad() to add whitespace

Examples

```
str_trim("  String with trailing and leading white space\t")
str_trim("\n\nString with trailing and leading white space\n\n")

str_squish("  String with trailing,  middle, and leading white space\t")
str_squish("\n\nString with excess,  trailing and leading white   space\n\n")
```

---

str_trunc *Truncate a string to maximum width*

Description

Truncate a string to a fixed of characters, so that `str_length(str_trunc(x, n))` is always less than or equal to `n`.

Usage

```
str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`width`	Maximum width of string.
`side, ellipsis`	Location and content of ellipsis that indicates content has been removed.

Value

A character vector the same length as `string`.

See Also

str_pad() to increase the minimum width of a string.

Examples

```
x <- "This string is moderately long"
rbind(
  str_trunc(x, 20, "right"),
  str_trunc(x, 20, "left"),
  str_trunc(x, 20, "center")
)
```

`str_unique` *Remove duplicated strings*

Description

`str_unique()` removes duplicated values, with optional control over how duplication is measured.

Usage

```
str_unique(string, locale = "en", ignore_case = FALSE, ...)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`locale`	Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that default behaviour is consistent across platforms.
`ignore_case`	Ignore case when comparing strings?
`...`	Other options used to control collation. Passed on to `stringi::stri_opts_collator()`.

Value

A character vector, usually shorter than `string`.

See Also

`unique()`, `stringi::stri_unique()` which this function wraps.

Examples

```
str_unique(c("a", "b", "c", "b", "a"))

str_unique(c("a", "b", "c", "B", "A"))
str_unique(c("a", "b", "c", "B", "A"), ignore_case = TRUE)

# Use ... to pass additional arguments to stri_unique()
str_unique(c("motley", "mötley", "pinguino", "pingüino"))
str_unique(c("motley", "mötley", "pinguino", "pingüino"), strength = 1)
```

`str_view` *View strings and matches*

Description

`str_view()` is used to print the underlying representation of a string and to see how a `pattern` matches.

Matches are surrounded by <> and unusual whitespace (i.e. all whitespace apart from `" "` and `"\n"`) are surrounded by `{}` and escaped. Where possible, matches and unusual whitespace are coloured blue and `NA`s red.

Usage

```
str_view(
  string,
  pattern = NULL,
  match = TRUE,
  html = FALSE,
  use_escapes = FALSE
)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. You can not match boundaries, including `""`, with this function.
`match`	If `pattern` is supplied, which elements should be shown? • `TRUE`, the default, shows only elements that match the pattern. • `NA` shows all elements. • `FALSE` shows only elements that don't match the pattern. If `pattern` is not supplied, all elements are always shown.
`html`	Use HTML output? If `TRUE` will create an HTML widget; if `FALSE` will style using ANSI escapes.
`use_escapes`	If `TRUE`, all non-ASCII characters will be rendered with unicode escapes. This is useful to see exactly what underlying values are stored in the string.

Value

If `html = FALSE`, a character vector of class `stringr_view` containing the elements selected by `match`, which prints one element per line. If `html = TRUE`, an HTML widget.

Examples

```
# Show special characters
str_view(c("\"\\", "\\\\\\", "fgh", NA, "NA"))

# A non-breaking space looks like a regular space:
nbsp <- "Hi\u00A0you"
nbsp
# But it doesn't behave like one:
str_detect(nbsp, " ")
# So str_view() brings it to your attention with a blue background
str_view(nbsp)

# You can also use escapes to see all non-ASCII characters
str_view(nbsp, use_escapes = TRUE)

# Supply a pattern to see where it matches
str_view(c("abc", "def", "fghi"), "[aeiou]")
str_view(c("abc", "def", "fghi"), "^")
str_view(c("abc", "def", "fghi"), "..")

# By default, only matching strings will be shown
str_view(c("abc", "def", "fghi"), "e")
# but you can show all:
str_view(c("abc", "def", "fghi"), "e", match = NA)
# or just those that don't match:
str_view(c("abc", "def", "fghi"), "e", match = FALSE)
```

`str_which` *Find matching indices*

Description

`str_which()` returns the indices of `string` where there's at least one match to `pattern`. It's a wrapper around `which(str_detect(x, pattern))`, and is equivalent to `grep(pattern, x)`.

Usage

```
str_which(string, pattern, negate = FALSE)
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`pattern`	Pattern to look for. The default interpretation is a regular expression, as described in stringr's regular expressions article. Use `regex()` for finer control of the matching behaviour. Match a fixed string (i.e. by comparing only bytes), using `fixed()`. This is fast, but approximate. Generally, for matching human text, you'll want `coll()` which respects character matching rules for the specified locale. You can not match boundaries, including `""`, with this function.

negate	If TRUE, inverts the resulting boolean vector.

Value

An integer vector, usually smaller than string.

Examples

```
fruit <- c("apple", "banana", "pear", "pineapple")
str_which(fruit, "a")

# Elements that don't match
str_which(fruit, "^p", negate = TRUE)

# Missings never match
str_which(c("a", NA, "b"), ".")
```

---

str_wrap *Wrap words into nicely formatted paragraphs*

Description

Wrap words into paragraphs, minimizing the "raggedness" of the lines (i.e. the variation in length line) using the Knuth-Plass algorithm.

Usage

```
str_wrap(string, width = 80, indent = 0, exdent = 0, whitespace_only = TRUE)
```

Arguments

string	Input vector. Either a character vector, or something coercible to one.
width	Positive integer giving target line width (in number of characters). A width less than or equal to 1 will put each word on its own line.
indent, exdent	A non-negative integer giving the indent for the first line (indent) and all subsequent lines (exdent).
whitespace_only	A boolean. • If TRUE (the default) wrapping will only occur at whitespace. • If FALSE, can break on any non-word character (e.g. /, -).

Value

A character vector the same length as string.

See Also

stringi::stri_wrap() for the underlying implementation.

Examples

```
thanks_path <- file.path(R.home("doc"), "THANKS")
thanks <- str_c(readLines(thanks_path), collapse = "\n")
thanks <- word(thanks, 1, 3, fixed("\n\n"))
cat(str_wrap(thanks), "\n")
cat(str_wrap(thanks, width = 40), "\n")
cat(str_wrap(thanks, width = 60, indent = 2), "\n")
cat(str_wrap(thanks, width = 60, exdent = 2), "\n")
cat(str_wrap(thanks, width = 0, exdent = 2), "\n")
```

---

str_write_lines *Write a character vector as lines to a file*

Description

`str_write_lines()` writes each element of `string` followed by `sep`, including the last one, and converts the output to `encoding`. It is the counterpart of `str_read_lines()`.

Usage

```
str_write_lines(string, con, encoding = "UTF-8", sep = "\n")
```

Arguments

`string`	A character vector to write.
`con`	A file name or a connection opened in binary mode.
`encoding`	A single string giving the output encoding. `NULL` or `""` uses the current default encoding.
`sep`	A single string written after each element. Defaults to `"\n"` on all platforms.

Details

Missing values are an error. Strings marked with the `"bytes"` encoding are an error, as in `str_c()`. Characters that the output encoding cannot represent are written as that encoding's substitution character, with a warning.

Value

`string`, invisibly and unchanged.

See Also

`str_read_lines()`; `stringi::stri_write_lines()`, which writes an empty file instead of signalling an error for missing values.

Examples

```
path <- tempfile()
str_write_lines(c("first", "second"), path)
str_read_lines(path, encoding = "UTF-8")
unlink(path)
```

`word` *Extract words from a sentence*

Description

Extract words from a sentence

Usage

```
word(string, start = 1L, end = start, sep = fixed(" "))
```

Arguments

`string`	Input vector. Either a character vector, or something coercible to one.
`start`, `end`	Pair of integer vectors giving range of words (inclusive) to extract. If negative, counts backwards from the last word. The default value select the first word.
`sep`	Separator between words. Defaults to single space.

Value

A character vector with the same length as `string`/`start`/`end`.

Examples

```
sentences <- c("Jane saw a cat", "Jane sat down")
word(sentences, 1)
word(sentences, 2)
word(sentences, -1)
word(sentences, 2, -1)

# Also vectorised over start and end
word(sentences[1], 1:3, -1)
word(sentences[1], 1, 1:4)

# Can define words by other separators
str <- 'abc.def..123.4568.999'
word(str, 1, sep = fixed('..'))
word(str, 2, sep = fixed('..'))
```

Index

∗ **datasets**
    stringr-data, 9

boundary *(*modifiers*)*, 7
boundary(), *12*, *16*, *22*, *32*

case, 3
charr_backend, 4
charr_chunking, 5
charr_chunks_per_worker *(*charr_chunking*)*, 5
charr_min_chunk *(*charr_chunking*)*, 5
charr_threads, 6
charr_threads(), *5*
coll *(*modifiers*)*, 7
coll(), *12*, *13*, *16*, *22*, *28*, *29*, *32*, *34*, *37*, *41*, *42*

emptyenv(), *19*

fixed *(*modifiers*)*, 7
fixed(), *12*, *13*, *15*, *16*, *22*, *28*, *29*, *32*, *34*, *37*, *41*, *42*
fruit *(*stringr-data*)*, 9

glue::glue(), *18*
glue::glue_data(), *18*
grep(), *37*

invert_match, 7

modifiers, 7

parallel::detectCores(), *6*
paste0(), *10*

regex *(*modifiers*)*, 7
regex(), *12*, *13*, *16*, *22*, *28*, *29*, *32*, *34*, *37*, *41*, *42*

sentences *(*stringr-data*)*, 9
str_c, 10
str_c(), *44*
str_conv, 11
str_count, 12
str_detect, 13
str_detect(), *34*
str_dup, 14
str_ends *(*str_starts*)*, 34
str_equal, 14
str_escape, 15
str_extract, 16
str_extract(), *22*, *24*, *36*
str_extract_all *(*str_extract*)*, 16
str_flatten, 17
str_flatten(), *10*
str_flatten_comma *(*str_flatten*)*, 17
str_glue, 18
str_glue_data *(*str_glue*)*, 18
str_ilike *(*str_like*)*, 21
str_length, 19
str_length(), *26*
str_like, 21
str_locate, 22
str_locate(), *12*, *35*
str_locate_all *(*str_locate*)*, 22
str_locate_all(), *7*, *12*
str_match, 23
str_match(), *16*
str_match_all *(*str_match*)*, 23
str_order, 24
str_pad, 26
str_pad(), *39*
str_rank *(*str_order*)*, 24
str_read_lines, 27
str_read_lines(), *44*
str_remove, 28
str_remove_all *(*str_remove*)*, 28
str_replace, 29
str_replace(), *28*

str_replace_all (str_replace), 29
str_replace_na, 30
str_replace_na(), 10, 29
str_reverse, 31
str_sort (str_order), 24
str_split, 32
str_split_1 (str_split), 32
str_split_fixed (str_split), 32
str_split_i (str_split), 32
str_squish (str_trim), 38
str_starts, 34
str_sub, 35
str_sub(), 20
str_sub<- (str_sub), 35
str_sub_all (str_sub), 35
str_subset, 36
str_subset(), 13
str_to_camel, 37
str_to_kebab (str_to_camel), 37
str_to_lower (case), 3
str_to_sentence (case), 3
str_to_snake (str_to_camel), 37
str_to_title (case), 3
str_to_upper (case), 3
str_trim, 38
str_trim(), 26
str_trunc, 39
str_trunc(), 26
str_unique, 40
str_view, 41
str_view_all (str_view), 41
str_which, 42
str_width (str_length), 19
str_width(), 26
str_wrap, 43
str_write_lines, 44
stringi::about_search_regex, 29, 34
stringi::stri_cmp_equiv(), 15
stringi::stri_count(), 12
stringi::stri_detect(), 13
stringi::stri_enc_list(), 11
stringi::stri_extract(), 16
stringi::stri_length(), 20
stringi::stri_locale_list(), 3, 8, 14, 25, 40
stringi::stri_locate(), 22
stringi::stri_match(), 24
stringi::stri_opts_brkiter(), 8
stringi::stri_opts_collator(), 8, 14, 25, 40
stringi::stri_opts_regex(), 8
stringi::stri_order(), 25
stringi::stri_read_lines(), 27
stringi::stri_replace(), 29
stringi::stri_reverse(), 31
stringi::stri_split(), 33
stringi::stri_sub(), 35
stringi::stri_subset(), 37
stringi::stri_unique(), 40
stringi::stri_wrap(), 43
stringi::stri_write_lines(), 44
stringr-data, 9

trim(), 19

unique(), 40

word, 45
words (stringr-data), 9