Package ‘stringr’

November 15, 2023

Title Simple, Consistent Wrappers for Common String Operations

Version 1.5.1

Description A consistent, simple and easy to use set of wrappers around the fantastic 'stringi' package. All function and argument names (and positions) are consistent, all functions deal with ``NA''s and zero length vectors in the same way, and the output from one function is easy to feed into the input of another.

License MIT + file LICENSE


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

Depends R (>= 3.6)

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

Suggests covr, dplyr, gt, htmltools, htmlwidgets, knitr, rmarkdown, testthat (>= 3.0.0), tibble

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

LazyData true

RoxygenNote 7.2.3

NeedsCompilation no

Author Hadley Wickham [aut, cre, cph], Posit Software, PBC [cph, fnd]

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2023-11-14 23:10:02 UTC
R topics documented:

- case ................................................................. 2
- invert_match ...................................................... 4
- modifiers .......................................................... 4
- stringr-data ....................................................... 6
- str_c ................................................................. 7
- str_conv ............................................................ 8
- str_count .......................................................... 9
- str_detect .......................................................... 10
- str_dup .............................................................. 11
- str_equal ........................................................... 11
- str_escape ......................................................... 12
- str_extract ......................................................... 13
- str_flatten ......................................................... 14
- str_glue ............................................................. 15
- str_length ........................................................ 16
- str_like ............................................................ 17
- str_locate ........................................................ 18
- str_match .......................................................... 19
- str_order .......................................................... 21
- str_pad .............................................................. 22
- str_remove ......................................................... 23
- str_replace ....................................................... 24
- str_replace_na .................................................... 25
- str_split .......................................................... 26
- str_starts ........................................................ 28
- str_sub ............................................................. 29
- str_subset ......................................................... 30
- str_trim ............................................................ 31
- str_trunc .......................................................... 32
- str_unique ........................................................ 33
- str_view ........................................................,... 34
- str_which .......................................................... 35
- str_wrap ........................................................... 36
- word ................................................................. 37

Index ......................................................... 39

---

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

```r
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

```r
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
```
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.
modifiers

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.
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.
stringr-data

Value

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

Examples

```r
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](https://github.com/dariusk/corpora). sentences is a collection of "Harvard sentences" used for standardised testing of voice.

Usage

sentences

fruit

words
str_c

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.

NULLs 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 dplyr::coalesce() or str_replace_na() 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.

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 vignette("regular-expressions"). 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(). An empty pattern, "", 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 vignette("regular-expressions"). 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(). An empty pattern, "", is equivalent to boundary("character").

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)

**Arguments**  
- **string**: Input vector. Either a character vector, or something coercible to one.  
- **times**: Number of times to duplicate each string.

**Value**  
A character vector the same length as string/times.

**Examples**  
fruit <- c("apple", "pear", "banana")  
str_dup(fruit, 2)  
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

\texttt{stringi::stri_cmp_equiv()} for the underlying implementation.

Examples

\begin{verbatim}
# 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)
\end{verbatim}
### Examples

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

---

### Description

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

### Usage

```r
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 vignette("regular-expressions"). 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()`. An empty pattern, ",", 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()`: an 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")

strExtract(shopping_list, "([a-z]+) of ([a-z]+)")
strExtract(shopping_list, "([a-z]+) of ([a-z]+)", group = 1)
strExtract(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**

```r
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**

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

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.

- `.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().

.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(.)} has {hp} hp")

str_length

**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).
**str_like**

**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", "\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 operator

**Description**

str_like() follows the conventions of the SQL LIKE operator:

- Must match the entire string.
- _ matches a single character (like .).
• % matches any number of characters (like .*).
• \% and \_ match literal % and _.
• The match is case insensitive by default.

Usage

str_like(string, pattern, ignore_case = TRUE)

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 Ignore case of matches? Defaults to TRUE to match the SQL LIKE operator.

Value

A logical vector the same length as string.

Examples

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

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 vignette("regular-expressions"). 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(). An empty pattern, "", 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 vignette("regular-expressions"). 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)
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**

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

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

```r
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 \texttt{str_length(str_pad(x, n))} is always greater than or equal to \texttt{n}.

Usage

\begin{verbatim}
str_pad(
    string,
    width,
    side = c("left", "right", "both"),
    pad = " ",
    use_width = TRUE
)
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{string} Input vector. Either a character vector, or something coercible to one.
  \item \texttt{width} Minimum width of padded strings.
  \item \texttt{side} Side on which padding character is added (left, right or both).
\end{itemize}
**str_remove**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pad</strong></td>
<td>Single padding character (default is a space).</td>
</tr>
<tr>
<td><strong>use_width</strong></td>
<td>If FALSE, use the length of the string instead of the width; see <code>str_width()</code>/<code>str_length()</code> for the difference.</td>
</tr>
</tbody>
</table>

**Value**

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

**See Also**

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

**Examples**

```r
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_remove**

Remove matched patterns

**Description**

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

**Usage**

```r
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 vignette("regular-expressions"). Use `regex()` for finer control of the matching behaviour.
str_replace

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()`. An empty pattern, "", is equivalent to `boundary("character")`.

Value

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

See Also

`str_replace()` for the underlying implementation.

Examples

```r
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

```r
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.
- `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, which will be called once for each match (from right to left) and its return value will be used to replace the match.
str_replace_na

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

See Also
str_replace_na() to turn missing values into "NA": stri_replace() for the underlying implementation.

Examples
fruits <- c("one apple", "two pears", "three bananas")
str_replace(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", ], max = 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.

Examples

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

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\textsuperscript{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 vignette("regular-expressions"). 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()`. An empty pattern, "", 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**

`stri_split()` for the underlying implementation.

**Examples**

```r
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_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

```r
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

```r
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). 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`.
- **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`.

See Also

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

```r
hw <- "Hadley Wickham"

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

# Negative indices 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

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

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 vignette("regular-expressions"). 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(). An empty pattern, "", is equivalent to boundary("character").

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_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.
StrTrim

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")

StrTrunc

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.
str_unique

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")
)

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 str_unique()
str_unique(c("motley", "mötley", "pinguino", "pingüino"))
str_unique(c("motley", "mötley", "pinguino", "pingüino"), strength = 1)

---

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 NAs red.

Usage

```r
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 vignette("regular-expressions"). 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()`. An empty pattern, "%", is equivalent to `boundary("character")`.
- **match**: If pattern is supplied, which elements should be shown?
  - TRUE, the default, shows only elements that match the pattern.
  - NA shows all elements.
str_which

- 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.

Examples

```r
# 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

```r
str_which(string, pattern, negate = FALSE)
```
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).
word

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")

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, 6
boundary (modifiers), 4
boundary(), 9, 10, 13, 19, 24, 27, 31, 34, 36
case, 2
coll (modifiers), 4
coll(), 9, 10, 13, 19, 24, 27, 28, 31, 34, 36
dplyr::coalesce(), 7
emptyenv(), 16
fixed (modifiers), 4
fixed(), 9, 10, 13, 19, 24, 27, 28, 31, 34, 36
fruit (stringr-data), 6

str_escape, 12
str_extract, 13
str_extract(), 19, 20, 30
str_extract_all (str_extract), 13
str_flatten, 14
str_flatten(), 7
str_flatten_comma (str_flatten), 14
str_glue, 15
str_glue_data (str_glue), 15
str_length, 16
str_length(), 23
str_like, 17
str_locate, 18
str_locate(), 9
str_locate_all (str_locate), 18
str_locate_all(), 4, 9
str_match, 19
str_match(), 13
str_match_all (str_match), 19
str_order, 21
str_pad, 22
str_pad(), 32, 33
str_rank (str_order), 21
str_remove, 23
str_remove_all (str_remove), 23
str_replace, 24
str_replace(), 24
str_replace_all (str_replace), 24
str_replace_na, 25
str_replace_na(), 7, 25
str_sort (str_order), 21
str_sort(), 26
str_split, 26
str_split_1 (str_split), 26
str_split_fixed (str_split), 26
str_split_i (str_split), 26
str_squish (str_trim), 31
str_starts, 28
str_sub, 29
str_sub(), 16
str_sub<-(str_sub), 29
str_sub_all(str_sub), 29
str_subset, 30
str_subset(), 10
str_to_lower(case), 2
str_to_sentence(case), 2
str_to_title(case), 2
str_to_upper(case), 2
str_trim, 31
str_trim(), 23
str_trunc, 32
str_trunc(), 23
str_unique, 33
str_view, 34
str_view_all(str_view), 34
str_which, 35
str_width(str_length), 16
str_width(), 23
str_wrap, 36
stri_replace(), 25
stri_split(), 27
stringi::about_search_regex, 24, 28
stringi::stri_cmp_equiv(), 12
stringi::stri_count(), 9
stringi::stri_detect(), 10
stringi::stri_enc_list(), 8
stringi::stri_extract(), 13
stringi::stri_length(), 17
stringi::stri_locale_list(), 3, 5, 11, 21, 33
stringi::stri_locate(), 19
stringi::stri_match(), 20
stringi::stri_opts_brkiter(), 5
stringi::stri_opts_collator(), 5, 11, 21, 33
stringi::stri_opts_regex(), 5
stringi::stri_order(), 22
stringi::stri_sub(), 29
stringi::stri_subset(), 31
stringi::stri_unique(), 33
stringi::stri_wrap(), 37
stringr-data, 6
unique(), 33
word, 37
words(stringr-data), 6