Package ‘stringr’

October 14, 2022

Title Simple, Consistent Wrappers for Common String Operations
Version 1.4.1
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 GPL-2 | file LICENSE
BugReports https://github.com/tidyverse/stringr/issues
Depends R (>= 3.1)
Imports glue (>= 1.2.0), magrittr, stringi (>= 1.1.7)
Suggests covr, htmltools, htmlwidgets, knitr, rmarkdown, testthat
VignetteBuilder knitr
Encoding UTF-8
LazyData true
RoxygenNote 7.2.1
NeedsCompilation no
Author Hadley Wickham [aut, cre, cph],
        RStudio [cph, fnd]
Maintainer Hadley Wickham <hadley@rstudio.com>
Repository CRAN
Date/Publication 2022-08-20 22:10:02 UTC

R topics documented:

case ................................................................. 2
invert_match ......................................................... 3
case

Convert case of a string.

Description

Convert case of a string.

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")
invert_match

Arguments

- **string**: String to modify
- **locale**: Locale to use for translations. Defaults to "en" (English) to ensure consistent default ordering across platforms.

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

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**

- **fixed** Compare literal bytes in the string. This is very fast, but not usually what you want for non-ASCII character sets.
- **coll** Compare strings respecting standard collation rules.
- **regex** The default. Uses ICU regular expressions.
- **boundary** Match boundaries between things.

**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?
- **locale** Locale to use for comparisons. See `stringi::stri_locale_list()` for all possible options. Defaults to "en" (English) to ensure that the default collation 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.

See Also

str_wrap() for breaking text to form paragraphs
stringi::stringi-search-boundaries for more detail on the various boundaries

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

str_extract_all("a\n\nb\nc", "a")
str_extract_all("a\n\nb\nc", regex("a", dotall = TRUE))
str_c

Description

Joins two or more vectors element-wise into a single character vector, optionally inserting sep between input vectors. If collapse is not NULL, it will be inserted between elements of the result, returning a character vector of length 1.

str_c

Join multiple strings into a single string.
Usage

\texttt{str\_c(..., sep = \"\", collapse = NULL)}

Arguments

\texttt{...}  
One or more character vectors. Zero length arguments are removed. Short arguments are recycled to the length of the longest.

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 \texttt{str\_replace\_na()} to convert \texttt{NA} to "NA"

\texttt{sep}  
String to insert between input vectors.

\texttt{collapse}  
Optional string used to combine input vectors into single string.

Details

To understand how \texttt{str\_c} works, you need to imagine that you are building up a matrix of strings. Each input argument forms a column, and is expanded to the length of the longest argument, using the usual recycling rules. The \texttt{sep} string is inserted between each column. If \texttt{collapse} is \texttt{NULL} each row is collapsed into a single string. If non-\texttt{NULL} that string is inserted at the end of each row, and the entire matrix collapsed to a single string.

Value

If \texttt{collapse = NULL} (the default) a character vector with length equal to the longest input string. If \texttt{collapse} is non-\texttt{NULL}, a character vector of length 1.

See Also

\texttt{paste()} for equivalent base R functionality, and \texttt{stringi::stri\_join()} which this function wraps

Examples

\texttt{str\_c("Letter: ", letters)}
\texttt{str\_c("Letter", letters, sep = ": ")}
\texttt{str\_c(letters, ", is for", ": "..."\))}
\texttt{str\_c(letters[-26], " comes before ", letters[-1])}

\texttt{str\_c(letters, collapse = ")")}
\texttt{str\_c(letters, collapse = ", "\))}

# Missing inputs give missing outputs
\texttt{str\_c(c("a", NA, "b"), ": -d")}

# Use \texttt{str\_replace\_NA} to display literal NAs:
\texttt{str\_c(str\_replace\_na(c("a", NA, "b")), ": -d")}
**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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String to re-encode.</td>
</tr>
<tr>
<td>encoding</td>
<td>Name of encoding. See <code>stringi::stri_enc_list()</code> for a complete list.</td>
</tr>
</tbody>
</table>

**Examples**

```r
# 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 the number of matches in a string.*

**Description**
Vectorised over `string` and `pattern`.

**Usage**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Input vector. Either a character vector, or something coercible to one.</td>
</tr>
<tr>
<td>pattern</td>
<td>Pattern to look for. The default interpretation is a regular expression, as described in <code>stringi::stringi-search-regex</code>. Control options with <code>regex()</code>. Match a fixed string (i.e. by comparing only bytes), using <code>fixed()</code>. This is fast, but approximate. Generally, for matching human text, you’ll want <code>coll()</code> which respects character matching rules for the specified locale. Match character, word, line and sentence boundaries with <code>boundary()</code>. An empty pattern, &quot;&quot;&quot;, is equivalent to <code>boundary(&quot;character&quot;)</code>.</td>
</tr>
</tbody>
</table>
str_detect

Value
An integer vector.

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("."))

---

Description
Vectorised over string and pattern. Equivalent to grepl(pattern, x). See str_which() for an equivalent to grep(pattern, x).

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 stringi::stringi-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.
   Match character, word, line and sentence boundaries with boundary(). An empty pattern, ".", is equivalent to boundary("character").
negate If TRUE, return non-matching elements.

Value
A logical vector.
See Also

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

Examples

```r
fruit <- c("apple", "banana", "pear", "pinapple")
str_detect(fruit, "a")
str_detect(fruit, "a\$")
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 and concatenate strings within a character vector.**

**Description**

Vectorised over string and times.

**Usage**

`str_dup(string, times)`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Input character vector.</td>
</tr>
<tr>
<td>times</td>
<td>Number of times to duplicate each string.</td>
</tr>
</tbody>
</table>

**Value**

A character vector.

**Examples**

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

Extract matching patterns from a string.

Description

Vectorised over string and pattern.

Usage

str_extract(string, pattern)

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 stringi::stringi-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.

Match character, word, line and sentence boundaries with boundary(). An empty pattern, "", is equivalent to boundary("character").

simplify If FALSE, the default, returns a list of character vectors. If TRUE returns a character matrix.

Value

A character vector.

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")

# 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**

Flatten a string

**Usage**

```r
str_flatten(string, collapse = "")
```

**Arguments**

- `string`: Character to flatten
- `collapse`: String to insert between each piece

**Value**

A character vector of length 1

**Examples**

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

---

### str_glue

**Format and interpolate a string with glue**

**Description**

These functions are wrappers around `glue::glue()` and `glue::glue_data()`, which provide a powerful and elegant syntax for interpolating strings. These wrappers provide a small set of the full options. Use the functions 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.

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

The length of a string.
Description

Technically this returns the number of "code points", in a string. One code point usually corresponds to one character, but not always. For example, an u with a umlaut might be represented as a single character or as the combination a u and an umlaut.

Usage

str_length(string)

Arguments

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

Value

A numeric vector giving number of characters (code points) in each element of the character vector. Missing string have missing length.

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))

# Two ways of representing a u with an umlaut
u1 <- "\u00fc"
u2 <- stringi::stri_trans_nfd(u1)
# The print the same:
u1
u2
# But have a different length
str_length(u1)
str_length(u2)
# Even though they have the same number of characters
str_count(u1)
str_count(u2)

str_locate

Locate the position of patterns in a string.

Description

Vectorised over string and pattern. If the match is of length 0, (e.g. from a special match like $) end will be one character less than start.
str_locate

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 stringi::stringi-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.

Match character, word, line and sentence boundaries with boundary(). An empty pattern, "", is equivalent to boundary("character").

Value

For str_locate, an integer matrix. First column gives start position of match, and second column gives end position. For str_locate_all a list of integer matrices.

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 matched groups from a string.

Description

Vectorised over string and pattern.

Usage

str_match(string, pattern)
str_match_all(string, pattern)

Arguments

string Input vector. Either a character vector, or something coercible to one.
pattern Pattern to look for, as defined by an ICU regular expression. See stringi::stringi-
search-regex for more details.

Value

For str_match, a character matrix. First column is the complete match, followed by one column
for each capture group. For str_match_all, a list of character matrices.

See Also

str_extract() to extract the complete match, stringi::stri_match() for the underlying im-
plementation.

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)

x <- c("<a> <b>", "<a> ", "<a>", ", NA)

str_match(x, "<.*?> <.*?>")
str_match_all(x, "<.*?>")

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


**str_order**

Order or sort a character vector.

### Description

Order or sort a character vector.

### Usage

```r
str_order(
  x,
  decreasing = FALSE,
  na_last = TRUE,
  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**: In which locale should the sorting occur? Defaults to the English. This ensures that code behaves the same way across platforms.
- **numeric**: If TRUE, will sort digits numerically, instead of as strings.
- **...**: Other options used to control sorting order. Passed on to `stringi::stri_opts_collator()`.

### See Also

`stringi::stri_order()` for the underlying implementation.
Examples

```r
str_order(letters)
str_sort(letters)

str_order(letters, locale = "haw")
str_sort(letters, locale = "haw")

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

---

**str_pad**

*Pad a string.*

**Description**

Vectorised over `string`, `width` and `pad`.

**Usage**

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

**Arguments**

- `string`: A character vector.
- `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).

**Value**

A character vector.

**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_remove

str_pad("a", 10, pad = c("-", "_", " "))

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

---

**str_remove**

Remove matched patterns in a string.

**Description**

Alias for `str_replace(string, pattern, "")`.

**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 `stringi::stringi-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.

  Match character, word, line and sentence boundaries with `boundary()`. An empty pattern, "", is equivalent to `boundary("character")`.

**Value**

A character vector.

**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 matched patterns in a string.

Description
Vectorised over string, pattern and replacement.

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::stringi-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.
replacement | A character vector of replacements. Should be either length one, or 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 ()).
         | To perform multiple replacements in each element of string, pass a named vector (c(pattern1 = replacement1)) to str_replace_all. Alternatively, pass a function to replacement: it will be called once for each match and its return value will be used to replace the match.
         | To replace the complete string with NA, use replacement = NA_character_.

Value
A character vector.

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]", "--")
str_replace_all(fruits, "[aeiou]", toupper)
str_replace_all(fruits, "b", NA_character_)
str_replace(fruits, "([aeiou])", "")
str_replace(fruits, "([aeiou])", "\1\1")
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**
Vectorised over string and pattern.

**Usage**

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

**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 \texttt{stringi::stringi-search-regex}. Control options with \texttt{regex()}.
  - Match a fixed string (i.e. by comparing only bytes), using \texttt{fixed()}. This is fast, but approximate. Generally, for matching human text, you’ll want \texttt{coll()} which respects character matching rules for the specified locale.
  - Match character, word, line and sentence boundaries with \texttt{boundary()}. An empty pattern, "", is equivalent to \texttt{boundary("character")}.
- `n`: number of pieces to return. Default (Inf) uses all possible split positions.
  - For \texttt{str_split_fixed}, if \texttt{n} is greater than the number of pieces, the result will be padded with empty strings.
- `simplify`: If \texttt{FALSE}, the default, returns a list of character vectors. If \texttt{TRUE} returns a character matrix.

**Value**
For \texttt{str_split_fixed}, a character matrix with \texttt{n} columns. For \texttt{str_split}, a list of character vectors.

**See Also**

- \texttt{stri_split()} for the underlying implementation.

**Examples**

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

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

# 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_starts**  
Detect the presence or absence of a pattern at the beginning or end of a string.

### Description

Vectorised over string and pattern.

### Usage

```r
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::stringi-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, return non-matching elements.

### Value

A logical vector.

### See Also

- `str_detect()` which this function wraps when pattern is regex.
str_sub

Examples

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

Description

str_sub will recycle all arguments to be the same length as the longest argument. If any arguments are of length 0, the output will be a zero length character vector.

Usage

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

Arguments

- `string`: input character vector.
- `start, end`: Two integer vectors. `start` gives the position of the first character (defaults to first), `end` gives the position of the last (defaults to last character). Alternatively, pass a two-column matrix to `start`. Negative values count backwards from the last character.
- `omit_na`: Single logical value. If `TRUE`, missing values in any of the arguments provided will result in an unchanged input.
- `value`: replacement string

Details

Substrings are inclusive - they include the characters at both start and end positions. `str_sub(string, 1, -1)` will return the complete substring, from the first character to the last.

Value

A character vector of substring from `start` to `end` (inclusive). Will be length of longest input argument.

See Also

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

*Keep strings matching a pattern, or find positions.*

**Description**

`str_subset()` is a wrapper around `x[str_detect(x, pattern)]`, and is equivalent to `grep(pattern, x, value = TRUE)`. `str_which()` is a wrapper around `which(str_detect(x, pattern))`, and is equivalent to `grep(pattern, x)`. See `str_detect()` for an equivalent to `grepl(pattern, x)`.
str_subset

Usage

str_subset(string, pattern, negate = FALSE)
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 stringi::stringi-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.
   Match character, word, line and sentence boundaries with boundary(). An empty pattern, "", is equivalent to boundary("character").
negate If TRUE, return non-matching elements.

Details

Vectorised over string and pattern

Value

A character vector.

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_which(fruit, "a")

str_subset(fruit, "^a")
str_subset(fruit, "a$")
str_subset(fruit, "b")
str_subset(fruit, ["aeiou"])  
# Returns elements that do NOT match
str_subset(fruit, "^p", negate = TRUE)

# Missings never match
str_subset(c("a", NA, "b"), ".")
str_which(c("a", NA, "b"), ".")
**str_trim**

*Trim whitespace from a string*

**Description**

`str_trim()` removes whitespace from start and end of string; `str_squish()` also reduces repeated whitespace inside a string.

**Usage**

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

**Arguments**

- `string`: A character vector.
- `side`: Side on which to remove whitespace (left, right or both).

**Value**

A character vector.

**See Also**

`str_pad()` to add whitespace

**Examples**

```r
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 character string.*

**Description**

Truncate a character string.

**Usage**

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

- **string**: A character vector.
- **width**: Maximum width of string.
- **side, ellipsis**: Location and content of ellipsis that indicates content has been removed.

See Also

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

Examples

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

---

**str_view**

View HTML rendering of regular expression match.

Description

`str_view` shows the first match; `str_view_all` shows all the matches. To build regular expressions interactively, check out the RegExplain RStudio addin.

Usage

```r
str_view(string, pattern, match = NA)
str_view_all(string, pattern, match = NA)
```

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::stringi-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.
  - Match character, word, line and sentence boundaries with `boundary()`. An empty pattern, "\n", is equivalent to `boundary("character")`.
- **match**: If TRUE, shows only strings that match the pattern. If FALSE, shows only the strings that don’t match the pattern. Otherwise (the default, NA) displays both matches and non-matches.
str_view

Examples

str_view(c("abc", "def", "fgh"), "[aeiou]")
str_view(c("abc", "def", "fgh"), "^")
str_view(c("abc", "def", "fgh"), ".")

# Show all matches with str_view_all
str_view_all(c("abc", "def", "fgh"), "d|e")

# Use match to control what is shown
str_view(c("abc", "def", "fgh"), "d|e")
str_view(c("abc", "def", "fgh"), "d|e", match = TRUE)
str_view(c("abc", "def", "fgh"), "d|e", match = FALSE)

---

str_wrap

Wrap strings into nicely formatted paragraphs.

Description

This is a wrapper around stringi::stri_wrap() which implements the Knuth-Plass paragraph wrapping algorithm.

Usage

str_wrap(string, width = 80, indent = 0, exdent = 0)

Arguments

string character vector of strings to reformat.
width positive integer giving target line width in characters. A width less than or equal to 1 will put each word on its own line.
indent non-negative integer giving indentation of first line in each paragraph
exdent non-negative integer giving indentation of following lines in each paragraph

Value

A character vector of re-wrapped strings.

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

\textit{Extract words from a sentence.}

\textbf{Description}

Extract words from a sentence.

\textbf{Usage}

\begin{verbatim}
word(string, start = 1L, end = start, sep = fixed(" "))
\end{verbatim}

\textbf{Arguments}

- \texttt{string} \hspace{1cm} input character vector.
- \texttt{start} \hspace{1cm} integer vector giving position of first word to extract. Defaults to first word. If negative, counts backwards from last character.
- \texttt{end} \hspace{1cm} integer vector giving position of last word to extract. Defaults to first word. If negative, counts backwards from last character.
- \texttt{sep} \hspace{1cm} separator between words. Defaults to single space.

\textbf{Value}

character vector of words from \texttt{start} to \texttt{end} (inclusive). Will be length of longest input argument.

\textbf{Examples}

\begin{verbatim}
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('..'))
\end{verbatim}
Index

* datasets
  stringr-data, 6

boundary (modifiers), 4
boundary(), 8, 9, 11, 15, 19, 22, 26, 28

case, 2
coll (modifiers), 4
coll(), 8, 9, 11, 15, 19, 20, 22, 23, 26, 28

emptyenv(), 13

fixed (modifiers), 4
fixed(), 8, 9, 11, 15, 19, 20, 22, 23, 26, 28

fruit (stringr-data), 6

glue::glue(), 12
glue::glue_data(), 12
grep(), 26

invert_match, 3

modifiers, 4

paste(), 7

regex (modifiers), 4
regex(), 8, 9, 11, 15, 19, 20, 22, 23, 26, 28

sentences (stringr-data), 6
str_c, 6
str_conv, 8
str_count, 8
str_detect, 9
str_detect(), 23, 25
str_dup, 10
str_ends (str Starts), 23
str_extract, 11
str_extract(), 15, 16
str_extract_all (str_extract), 11
str_flatten, 12

str_glue, 12
str_glue_data (str_glue), 12
str_length, 13
str_locate, 14
str_locate(), 9
str_locate_all (str locate), 14
str_locate_all(), 3, 9
str_match, 16
str_match(), 11
str_match_all (str_match), 16
str_order, 17
str_pad, 18
str_pad(), 27, 28
str_remove, 19
str_remove_all (str_remove), 19
str_replace, 20
str_replace(), 19
str_replace_all (str_replace), 20
str_replace_na, 21
str_replace_na(), 7, 20
str_sort (str_order), 17
str_split, 22
str_split_fixed (str_split), 22
str_squish (str_trim), 27
str_starts, 23
str_sub, 24
str_sub<- (str_sub), 24
str_subset, 25
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, 27
str_trim(), 18
str_trunc, 27
str_trunc(), 18
str_view, 28
str_view_all (str_view), 28
str_which(str_subset), 25
str_which(), 9
str_wrap, 29
str_wrap(), 5
stri_replace(), 20
stri_split(), 22
stringi::stri_count(), 9
stringi::stri_detect(), 10
stringi::stri_enc_list(), 8
stringi::stri_extract(), 11
stringi::stri_join(), 7
stringi::stri_length(), 14
stringi::stri_locale_list(), 4
stringi::stri_locate(), 15
stringi::stri_match(), 16
stringi::stri_opts_brkiter(), 4
stringi::stri_opts_collator(), 4, 17
stringi::stri_opts_regex(), 4
stringi::stri_order(), 17
stringi::stri_sub(), 24
stringi::stri_subset(), 26
stringi::stri_wrap(), 29
stringi::stringi-search-regex, 8, 9, 11, 15, 16, 19, 20, 22, 23, 26, 28
stringr-data, 6

word, 30
words(stringr-data), 6