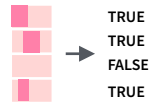


String manipulation with stringr : : CHEAT SHEET

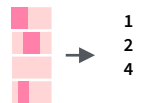


The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

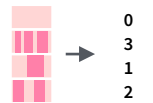
Detect Matches



str_detect(string, pattern) Detect the presence of a pattern match in a string. `str_detect(fruit, "a")`



str_which(string, pattern) Find the indexes of strings that contain a pattern match. `str_which(fruit, "a")`



str_count(string, pattern) Count the number of matches in a string. `str_count(fruit, "a")`

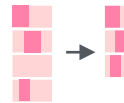


str_locate(string, pattern) Locate the positions of pattern matches in a string. Also **str_locate_all**. `str_locate(fruit, "a")`

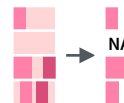
Subset Strings



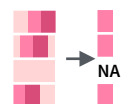
str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. `str_sub(fruit, 1, 3); str_sub(fruit, -2)`



str_subset(string, pattern) Return only the strings that contain a pattern match. `str_subset(fruit, "b")`

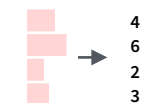


str_extract(string, pattern) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. `str_extract(fruit, "[aeiou]")`

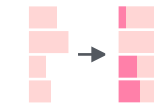


str_match(string, pattern) Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also **str_match_all**. `str_match(sentences, "(a|the) ([^]+)")`

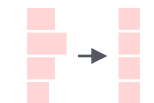
Manage Lengths



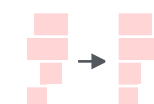
str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). `str_length(fruit)`



str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. `str_pad(fruit, 17)`



str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. `str_trunc(fruit, 3)`

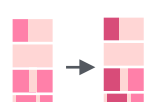


str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. `str_trim(fruit)`

Mutate Strings



str_sub() <- value. Replace substrings by identifying the substrings with `str_sub()` and assigning into the results. `str_sub(fruit, 1, 3) <- "str"`



str_replace(string, pattern, replacement) Replace the first matched pattern in each string. `str_replace(fruit, "a", "-")`



str_replace_all(string, pattern, replacement) Replace all matched patterns in each string. `str_replace_all(fruit, "a", "-")`

A STRING
↓
a string

str_to_lower(string, locale = "en")¹ Convert strings to lower case. `str_to_lower(sentences)`

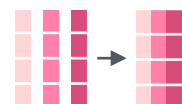
a string
↓
A STRING

str_to_upper(string, locale = "en")¹ Convert strings to upper case. `str_to_upper(sentences)`

a string
↓
A String

str_to_title(string, locale = "en")¹ Convert strings to title case. `str_to_title(sentences)`

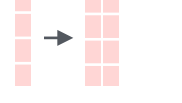
Join and Split



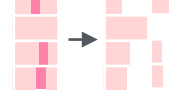
str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. `str_c(letters, LETTERS)`



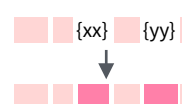
str_c(..., sep = "", collapse = NULL) Collapse a vector of strings into a single string. `str_c(letters, collapse = "")`



str_dup(string, times) Repeat strings times times. `str_dup(fruit, times = 2)`



str_split_fixed(string, pattern, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings. `str_split_fixed(fruit, " ", n=2)`



str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. `str_glue("Pi is {pi}")`



str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. `str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")`

Order Strings



str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...) ¹ Return the vector of indexes that sorts a character vector. `x[str_order(x)]`



str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...) ¹ Sort a character vector. `str_sort(x)`

Helpers

apple
banana
pear

str_conv(string, encoding) Override the encoding of a string. `str_conv(fruit, "ISO-8859-1")`

apple
banana
pear

str_view(string, pattern, match = NA) View HTML rendering of first regex match in each string. `str_view(fruit, "[aeiou]")`

str_view_all(string, pattern, match = NA) View HTML rendering of all regex matches. `str_view_all(fruit, "[aeiou]")`

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. `str_wrap(sentences, 20)`

Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed*.

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("" or '') or single quotes('').

Some characters cannot be represented directly in an R string. These must be represented as **special characters**, sequences of characters that have a specific meaning, e.g.

Special Character	Represents
<code>\\</code>	<code>\</code>
<code>\"</code>	"
<code>\\n</code>	new line

Run `?\"` to see a complete list

Because of this, whenever a `\` appears in a regular expression, you must write it as `\\` in the string that represents the regular expression.

Use `writeLines()` to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.
```

```
writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's, and/or to have `.` match everything including `\n`.
`str_detect("I", regex("i", TRUE))`

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). `str_detect("\u0130", fixed("i"))`

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow).
`str_detect("\u0130", coll("i", TRUE, locale = "tr"))`

boundary() Matches boundaries between characters, line_breaks, sentences, or words.
`str_split(sentences, boundary("word"))`

Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

string (type this)	regex (to mean this)	matches (which matches this)	example
	a (etc.)	a (etc.)	
<code>\\.</code>	<code>\.</code>	<code>.</code>	<code>see("a")</code> abc ABC 123 <code>.!?\()</code>
<code>\\!</code>	<code>\\!</code>	<code>!</code>	<code>see("\\.")</code> abc ABC 123 <code>.!?\()</code>
<code>\\?</code>	<code>\\?</code>	<code>?</code>	<code>see("\\!")</code> abc ABC 123 <code>.!?\()</code>
<code>\\\\</code>	<code>\\</code>	<code>\</code>	<code>see("\\?")</code> abc ABC 123 <code>.!?\()</code>
<code>\\(</code>	<code>\\(</code>	<code>(</code>	<code>see("\\\\")</code> abc ABC 123 <code>.!?\()</code>
<code>\\)</code>	<code>\\)</code>	<code>)</code>	<code>see("\\(")</code> abc ABC 123 <code>.!?\()</code>
<code>\\{</code>	<code>\\{</code>	<code>{</code>	<code>see("\\}")</code> abc ABC 123 <code>.!?\()</code>
<code>\\}</code>	<code>\\}</code>	<code>}</code>	<code>see("\\{")</code> abc ABC 123 <code>.!?\()</code>
<code>\\n</code>	<code>\\n</code>	new line (return)	<code>see("\\n")</code> abc ABC 123 <code>.!?\()</code>
<code>\\t</code>	<code>\\t</code>	tab	<code>see("\\t")</code> abc ABC 123 <code>.!?\()</code>
<code>\\s</code>	<code>\\s</code>	any whitespace (S for non-whitespaces)	<code>see("\\s")</code> abc ABC 123 <code>.!?\()</code>
<code>\\d</code>	<code>\\d</code>	any digit (D for non-digits)	<code>see("\\d")</code> abc ABC 123 <code>.!?\()</code>
<code>\\w</code>	<code>\\w</code>	any word character (W for non-word chars)	<code>see("\\w")</code> abc ABC 123 <code>.!?\()</code>
<code>\\b</code>	<code>\\b</code>	word boundaries	<code>see("\\b")</code> abc ABC 123 <code>.!?\()</code>
	[:digit:] ¹	digits	<code>see("[:digit:]")</code> abc ABC 123 <code>.!?\()</code>
	[:alpha:] ¹	letters	<code>see("[:alpha:]")</code> abc ABC 123 <code>.!?\()</code>
	[:lower:] ¹	lowercase letters	<code>see("[:lower:]")</code> abc ABC 123 <code>.!?\()</code>
	[:upper:] ¹	uppercase letters	<code>see("[:upper:]")</code> abc ABC 123 <code>.!?\()</code>
	[:alnum:] ¹	letters and numbers	<code>see("[:alnum:]")</code> abc ABC 123 <code>.!?\()</code>
	[:punct:] ¹	punctuation	<code>see("[:punct:]")</code> abc ABC 123 <code>.!?\()</code>
	[:graph:] ¹	letters, numbers, and punctuation	<code>see("[:graph:]")</code> abc ABC 123 <code>.!?\()</code>
	[:space:] ¹	space characters (i.e. <code>\s</code>)	<code>see("[:space:]")</code> abc ABC 123 <code>.!?\()</code>
	[:blank:] ¹	space and tab (but not new line)	<code>see("[:blank:]")</code> abc ABC 123 <code>.!?\()</code>
	.	every character except a new line	<code>see(".")</code> abc ABC 123 <code>.!?\()</code>

¹ Many base R functions require classes to be wrapped in a second set of `[]`, e.g. `[:digit:]`

[:space:]
new line

[:blank:]
space
tab

[:graph:]

[:punct:]

. , : ; ? ! \ | / ` = * + - ^
_ ~ " ' [] { } () < > @ # \$

[:alnum:]

[:digit:]

0 1 2 3 4 5 6 7 8 9

[:alpha:]

[:lower:]

a b c d e f
g h i j k l
m n o p q r
s t u v w x
z

[:upper:]

A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Z

ALTERNATES

`alt <- function(rx) str_view_all("abcde", rx)`

regex	matches	example
<code>ab d</code>	or	<code>alt("ab d")</code> abcde
<code>[abe]</code>	one of	<code>alt("[abe]")</code> abcde
<code>^[abe]</code>	anything but	<code>alt("^[abe]")</code> abcde
<code>[a-c]</code>	range	<code>alt("[a-c]")</code> abcde

ANCHORS

`anchor <- function(rx) str_view_all("aaa", rx)`

regex	matches	example
<code>^a</code>	start of string	<code>anchor("^a")</code> aaa
<code>a\$</code>	end of string	<code>anchor("a\$")</code> aaa

LOOK AROUNDS

`look <- function(rx) str_view_all("bacad", rx)`

regex	matches	example
<code>a(=?c)</code>	followed by	<code>look("a(=?c)")</code> bacad
<code>a(?!c)</code>	not followed by	<code>look("a(?!c)")</code> bacad
<code>(?<=b)a</code>	preceded by	<code>look("(?<=b)a")</code> bacad
<code>(?<!b)a</code>	not preceded by	<code>look("(?<!b)a")</code> bacad

QUANTIFIERS

`quant <- function(rx) str_view_all("a.aa.aaa", rx)`

regex	matches	example
<code>a?</code>	zero or one	<code>quant("a?")</code> a.aa.aaa
<code>a*</code>	zero or more	<code>quant("a*")</code> a.aa.aaa
<code>a+</code>	one or more	<code>quant("a+")</code> a.aa.aaa
<code>a{n}</code>	exactly n	<code>quant("a{2}")</code> a.aa.aaa
<code>a{n,}</code>	n or more	<code>quant("a{2,}")</code> a.aa.aaa
<code>a{n,m}</code>	between n and m	<code>quant("a{2,4}")</code> a.aa.aaa

GROUPS

`ref <- function(rx) str_view_all("abbaab", rx)`

Use parentheses to set precedent (order of evaluation) and create groups

regex	matches	example
<code>(ab d)e</code>	sets precedence	<code>alt("(ab d)e")</code> abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regex (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))
<code>\\1</code>	<code>\\1</code> (etc.)	first () group, etc.	<code>ref("(a)(b)\\2\\1")</code> abbaab

