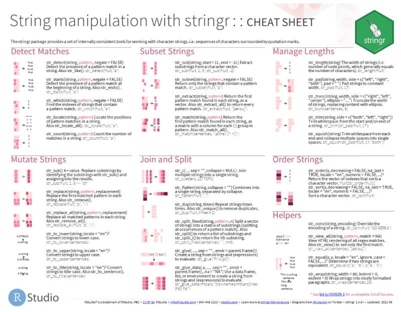
posit

String manipulation with stringr :: Cheatsheet

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

```
library(stringr)
```

Download PDF



String manipulation with stringr : : CHEAT SHEET

Detect Matches

Subset Strings

Manage Lengths

Mutate Strings

Join and Split

Order Strings

Helpers

Translations (PDF)

- Portuguese
- Spanish
- Vietnamese

Detect Matches

- `str_detect(string, pattern, negate = FALSE)`: Detect the presence of a pattern match in a string. Also `str_like()`.

```
str_detect(fruit, "a")
```

- `str_starts(string, pattern, negate = FALSE)`: Detect the presence of a pattern match at the beginning of a string. Also `str_ends()`.

```
str_starts(fruit, "a")
```

- `str_which(string, pattern, negate = FALSE)`: Find the indexes of strings that contain a pattern match.

```
str_which(fruit, "a")
```

- `str_locate(string, pattern)`: Locate the positions of pattern matches in a string. Also `str_locate_all()`.

```
str_locate(fruit, "a")
```

- `str_count(string, pattern)`: Count the number of matches in a string.

```
str_count(fruit, "a")
```

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. Also `str_remove()`.

```
str_replace(fruit, "p", "-")
```

- `str_replace_all(string, pattern, replacement)`: Replace all matched patterns in each string. Also `str_remove_all()`.

```
str_replace_all(fruit, "p", "-")
```

- `str_to_lower(string, locale = "en")`[1]: Convert strings to lower case.

```
str_to_lower(sentences)
```

- `str_to_upper(string, locale = "en")`[1]: Convert strings to upper case.

```
str_to_upper(sentences)
```

- `str_to_title(string, locale = "en")` [1]: Convert strings to title case. Also `str_to_setence()`.

```
str_to_title(sentences)
```

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, negate = FALSE)`: Return only the strings that contain a pattern match.

```
str_subset(fruit, "p")
```

- `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) ([^ +])")
```

Join and Split

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

```
str_c(letters, LETTERS)
```

- `str_flatten(string, collapse = "")`: Combines into a single string, separated by collapse.

```
str_flatten(fruit, ", ")
```

- `str_dup(string, times)`: Repeat strings times times. Also `str_unique()` to remove duplicates.

```
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 and `str_split_i()` to return the ith substring.

```
str_split_fixed(sentences, " ", n = 3)
```

- `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")
```

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("left", "right", "both"), ellipsis = "...")`: Truncate the width of strings, replacing content with ellipsis.

```
str_trunc(sentences, 6)
```

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

```
str_trim(str_pad(fruit, 17))
```

- `str_squish(string)`: Trim white space from each end and collapse multiple spaces into single spaces.

```
str_squish(str_pad(fruit, 17, "both"))
```

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.

```
fruit[str_order(fruit)]
```

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

```
str_sort(fruit)
```

Helpers

- `str_conv(string, encoding)`: Override the encoding of a string.

```
str_conv(fruit, "ISO-8859-1")
```

- `str_view(string, pattern, match = NA)`: View HTML rendering of all regex matches. Also `str_view()` to see only the first match.

```
str_view(sentences, "[aeiou]")
```

- `str_equal(x, y, locale = "en", ignore_case = FALSE, ...)`[1]: Determine if two strings are equivalent.

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

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

```
str_wrap(sentences, 20)
```

[1] See http://bit.ly/ISO639-1 for a complete list of locales.

Regular Expressions

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

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 single quotes (`''`).

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

For example, `writeLines("\\.")` will be parsed as `\.`

and `writeLines("\\ is a backslash")` will be parsed as `\ 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 as end of strings, allow R comments within regexs, and/or to have `.` match everthing 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"))
```

Match Characters

```
see <- function(rx) str_view("abc ABC 123\t.!?\\(){}\n", rx)
```

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

string (type this)	regex (to mean this)	matches (which matches this)	example	example output (highlighted characters are in <>)
	`a (etc.)`	`a (etc.)`	`see("a")`	`<a>bc ABC 123\t.!?\(){}\n`
`\\.`	`\.`	`.`	`see("\\.")```	`abc ABC 123\t<.>!?\(){}\n`
`\\!`	`\!`	`!`	`see("\\!")`	`abc ABC 123\t.<!>?\(){}\n`
`\\?`	`\?`	`?`	`see("\\?")`	`abc ABC 123\t.!<?>\(){}\n`
`\\\\`	`\\`	`\`	`see("\\\\")`	`abc ABC 123\t.!?<\>(){}\n`
`\\(`	`\(`	`(`	`see("\\(")`	`abc ABC 123\t.!?\<(>){}\n`
`\\)`	`\)`	`)`	`see("\\)")`	`abc ABC 123\t.!?\(<)>{}\n`
`\\{`	`\{`	`{`	`see("\\{")`	`abc ABC 123\t.!?\()<{>}\n`

string (type this)	regex (to mean this)	matches (which matches this)	example	example output (highlighted characters are in <>)
\\}	\}	}	see("\\}")	abc ABC 123\t.!?\(){<}>\n
\\n	\n	new line (return)	see("\\n")	abc ABC 123\t.!?\(){}<\n>
\\t	\t	tab	see("\\t")	abc ABC 123<\t>.!?\(){}\n
\\s	\s	any whitespace (\S for non-whitespaces)	see("\\s")	abc< >ABC< >123<\t>.!?\(){}<\n>
\\d	\d	any digit (\D for non-digits)	see("\\d")	abc ABC <1><2><3>\t.!?\(){}\n
\\w	\w	any word character (\W for non-word characters)	see("\\w")	<a><b><c> <A><B><C> <1><2><3>\t.!?\(){}\n
\\b	\b	word boundaries	see("\\b")	<>abc<> <>ABC<> <>123<>\t.!?\(){}\n
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC <1><2><3>\t.!?\(){}\n
	[:alpha:] [1]	letters	see("[:alpha:]")	<a><b><c> <A><B><C> 123\t.!?\(){}\n
	[:lower:] [1]	lowercase letters	see("[:lower:]")	<a><b><c> ABC 123\t.!?\(){}\n

string (type this)	regex (to mean this)	matches (which matches this)	example	example output (highlighted characters are in <>)
	`[:upper:]`[1]	uppercase letters	`see("[:upper:]")`	`abc <A><B><C> 123\t.!?\(){}\n`
	`[:alnum:]`[1]	letters and numbers	`see("[:alnum:]")`	`<a><b><c> <A><B><C> <1><2><3>\t.!?\(){}\n`
	`[:punct:]`[1]	punctuation	`see("[:punct:]")`	`abc ABC 123\t<.><!><?><\><(><)><{><}>\n`
	`[:graph:]`[1]	letters, numbers, and punctuation	`see("[:graph:]")`	`<a><b><c> <A><B><C> <1><2><3>\t<.><!><?><\><(><)><{><}>\n`
	`[:space:]`[1]	space characters (i.e. `\s`)	`see("[:space:]")`	`abc< >ABC< >123<\t>.!?\(){}<\n>`
	`[:blank:]`[1]	space and tab (but not new line)	`see("[:blank:]")`	`abc< >ABC< >123<\t>.!?\(){}\n`
	`.`	every character except a new line	`see(".")`	`<a><b><c>< ><A><B><C>< ><1><2><3><\t><.><!><?><\><(><)><{><}><\n>`

Classes

- The `[:space:]` class includes new line, and the `[:blank:]` class
    - The `[:blank:]` class includes space and tab (`\t`)
- The `[:graph:]` class contains all non-space characters, including `[:punct:]`, `[:symbol:]`, `[:alnum:]`, `[:digit:]`, `[:alpha:]`, `[:lower:]`, and `[:upper:]`
    - `[:punct:]` contains punctuation: `. , : ; ? ! / * @ # - _ " [ ] { } ( )`

  - `[:symbol:]` contains symbols: `|` `` ` `` `=` `+` `^` `~` `<` `>` `$`
  - `[:alnum:]` contains alphanumeric characters, including `[:digit:]`, `[:alpha:]`, `[:lower:]`, and `[:upper:]`
    - `[:digit:]` contains the digits `0` through `9`
    - `[:alpha:]` contains letters, including `[:upper:]` and `[:lower:]`
      - `[:upper:]` contains uppercase letters and `[:lower:]` contains lowercase letters
- The regex `.` contains all characters in the above classes, except new line.

Alternates

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

Alternates

regexp	matches	example	example output (highlighted characters are in <>)
`ab|d`	or	`alt("ab|d")`	`<ab>c<d>e`
`[abe]`	one of	`alt("[abe]"`	`<a><b>cd<e>`
`[^abe]`	anything but	`alt("[^abe]")`	`ab<c><d>e`
`[a-c]`	range	`alt("[a-c]")`	`<a><b><c>de`

Anchors

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

Anchors

regexp | matches | example | example output
| | | (highlighted characters are in <>)

`^a` | start of string | `anchor("^a")` | `|` `|` `|` `<a>aa` `|`
`|` `|`

`a$` | end of string | `anchor("a$")` | `|` `|` `|` `aa<a>` `|`
`|` `|`

Look Arounds

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

Look arounds

regexp	matches	example	example output (highlighted characters are in <>)
`a(?=c)`	followed by	`look("a(?=c)")`	`b<a>cad`
`a(?!c)`	not followed by	`look("a(?!c)")`	`bac<a>d`
`(?<=b)a`	preceded by	`look("(?<=b)a")`	`b<a>cad`
`(?<!b)a`	not preceded by	`look("(?<!b)a")`	`bac<a>d`

Quantifiers

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

Quantifiers

regexp	matches	example	example output (highlighted characters are in <>)
`a?`	zero or one	`quant("a?")`	`<>.<a><>.<a><a><>.<a><a><a><>`
`a*`	zero or more	`quant("a*")`	`<>.<a><>.<aa><>.<aaa><>`
`a+`	one or more	`quant("a+")`	`.<a>.<aa>.<aaa>`
`a{n}`	exactly `n`	`quant("a{2}")`	`.a.<aa>.<aa>a`
`a{n, }`	`n` or more	`quant("a{2,}")`	`.a.<aa>.<aaa>`
`a{n, m}`	between `n` and `m`	`quant("a{2,4}")`	`.a.<aa>.<aaa>`

Groups

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

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

Groups

regexp	matches	example	example output (highlighted characters are in <>)
`(ab|d)e`	sets precedence	`alt("(ab|d)e")`	`abc<de>`

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

More groups

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as `ref("abba")`)	example output (highlighted characters are in <>)
`\\1`	`\1` (etc.)	first () group, etc.	`ref("(a)(b)\\2\\1")`	`<abba>ab`

Learn more at stringr.tidyverse.org.

Updated: 2025-01.

```
packageVersion("stringr")
```

```
[1] '1.5.1'
```