Package: stringmagic (via r-universe)

October 15, 2024

Type Package

Title Character String Operations and Interpolation, Magic Edition

Version 1.2.0

Imports Rcpp(>= 1.0.5), utils, stats

Suggests knitr, rmarkdown, data.table

LinkingTo Rcpp

Description Performs complex string operations compactly and efficiently. Supports string interpolation jointly with over 50 string operations. Also enhances regular string functions (like grep() and co). See an introduction at <https://lrberge.github.io/stringmagic/>.

License GPL (>= 2)

Encoding UTF-8

BugReports https://github.com/lrberge/stringmagic/issues

URL https://lrberge.github.io/stringmagic/, https://github.com/lrberge/stringmagic

VignetteBuilder knitr

RoxygenNote 7.3.1

Roxygen list(markdown = TRUE)

Repository https://lrberge.r-universe.dev

RemoteUrl https://github.com/lrberge/stringmagic

RemoteRef HEAD

RemoteSha e9df4fdd55b506b4aa045a1560bddbae7371a141

Contents

`cat_magic_alias` *Display messages using interpolated strings*

Description

Utilities to display messages using `string_magic` interpolation and operations to generate the message.

Usage

```
cat_magic_alias(
  .sep = "",
  .end = "",
  .width = FALSE,
  .leader = "",
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .namespace = NULL
)

message_magic_alias(
  .sep = "",
  .end = "\n",
  .width = "min(100, .sw)",
  .leader = "",
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .namespace = NULL
)

cat_magic(
  ...,
  .sep = "",
  .end = "",
  .width = FALSE,
  .leader = "",
  .envir = parent.frame(),
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .help = NULL,
  .namespace = NULL
)

catma(
  ...,
  .sep = "",
  .end = "",
  .width = FALSE,
  .leader = "",
  .envir = parent.frame(),
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .help = NULL,
  .namespace = NULL
)

message_magic(
  ...,
  .sep = "",
  .end = "\n",
  .width = NULL,
  .leader = "",
  .envir = parent.frame(),
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .help = NULL,
  .namespace = NULL
)
```

```
mema(
  ...,
  .sep = "",
  .end = "\n",
  .width = NULL,
  .leader = "",
  .envir = parent.frame(),
  .delim = c("{", "}"),
  .last = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .help = NULL,
  .namespace = NULL
)
```

Arguments

`.sep`	Character scalar, default is the empty string `""`. It is used to collapse all the elements in `...` before applying any operation.
`.end`	Character scalar, default is `""` (the empty string) for `cat_magic`, and `"\n"` (a newline) for `message_magic`. This string will be collated at the end of the message (a common alternative is `"\n"`).
`.width`	Can be 1) a positive integer, 2) a number in (0;1), 3) `FALSE` (default for `cat_magic`), or 4) `NULL` (default for `message_magic`). It represents the target width of the message on the user console. Newlines will be added *between words* to fit the target width. 1. positive integer: number of characters 2. number (0;1): fraction of the screen 3. `FALSE`: does not add newlines 4. `NULL`: the min between 100 characters and 90% of the screen width Note that you can use the special variable `.sw` to refer to the screen width. Hence the value `NULL` is equivalent to using `min(100, 0.9*.sw)` (which can be passed as a character string).
`.leader`	Character scalar, default is `TRUE`. Only used if argument `.width` is not `FALSE`. Whether to add a leading character string right after the extra new lines.
`.delim`	Character vector of length 1 or 2. Default is `c("{", "}")`. Defines the opening and the closing delimiters for interpolation. If of length 1, it must be of the form: 1) the opening delimiter, 2) a single space, 3) the closing delimiter. Ex: `".[ ]"` is equivalent to `c(".[", "]")`. The default value is equivalent to `"{ }"`. [ ]: R:%20 [", "]: R:%22,%20%22
`.last`	Character scalar, a function, or `NULL` (default). If provided and character: it must be an `string_magic` chain of operations of the form `"'arg1'op1, op2, etc"`. All these operations are applied just before returning the vector. If a function, it will be applied to the resulting vector.

`.trigger`	Logical, default is `TRUE`. If `FALSE`, this function is not run. Can be useful in debugging situations where we want conditional evaluations.
`.check`	Logical scalar, default is `TRUE`. Whether to enable error-handling (i.e. human readable error messages). Without error-handling you can save something of the order of 40us. Useful only in long loops.
`.namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`.
`...`	Character scalars that will be collapsed with the argument `sep`. Note that named arguments are used for substitution. To interpolate, you can use `"{x}"` within each character string to insert the value of `x` in the string. You can add string operations in each `"{}"` instance with the syntax `"'arg'op ? x"` (resp. `"'arg'op ! x"`) to apply the operation `'op'` with the argument `'arg'` to `x` (resp. the verbatim of `x`). Otherwise, what to say? Ah, nesting is enabled, and since there's over 50 operators, it's a bit complicated to sort you out in this small space. Use the argument `.help = "keyword"` (or `.help = TRUE`) to obtain a selective help from the main documentation. Note that in interpolations you have access to the special variables: `.now` and `.date` to get the current time; and the special function `.now("format")` to format the time. Ex: `.now('%Y-%m %H:%m')`.
`.envir`	An environment used to evaluate the variables in `"{}"`. By default the variables are evaluated using the environment from where the function is called or using the named arguments passed to the function.
`.help`	Character scalar or `TRUE`, default is `NULL`. This argument is used to generate a dynamic help on the console. If `TRUE`, the user can select which topic to read from the main documentation, with the possibility to search for keywords and navigate the help pages. If a character scalar, then a regex search is perfomed on the main documentation and any section containining a match is displayed. The user can easily navigate across matches.

Details

These functions are `base::cat()`/`message()` wrappers aroung `string_magic()`. There is one notable difference with respect to `cat/message`. It's the ability to add newlines after words for the message to fit a target width. This is controlled with the argument `.width`. This is active by default for `message_magic` (default is `.width = NULL` which leads to the minimum betwen 100 characters and 90% of the screen width).

You can very easily change the default values with the alias generators `cat_magic_alias` and `message_magic_alias`.
`[Advanced]` A note for package developers who would use these functions **and** also use custom `string_magic` operations created with `string_magic_register_fun()` or `string_magic_register_ops()`. To ensure forward compatibility the new operations created should be defined in the package namespace (see the *ad hoc* section in `string_magic_register_fun()` help). To access these operators in their specific namespaces, you must use an alias with `cat_magic_alias` or `message_magic_alias`

with the argument `.namespace = "myPackageName"` (to avoid having to provide the `.namespace` argument repeatedly).

Value

The functions `cat_magic()` and `message_magic()` do not return anything, they simply print on the console.

The function `cat_magic_alis()` returns a function behaving identically to `cat_magic()` but for which the default values have been altered. Same for `message_magic_alias()`.

Functions

- `cat_magic_alias()`: Create an alias of `cat_magic` with custom defaults
- `message_magic_alias()`: Create an alias of `message_magic` with custom defaults
- `catma()`: Alias to `cat_magic`
- `message_magic()`: Display messages using interpolated strings
- `mema()`: Alias to `message_magic`

See Also

Other tools with aliases: `string_clean_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
start = Sys.time()
Sys.sleep(0.05)
message_magic("This example has run in {difftime ? start}.")

cat_magic("Let's write a very long message to illustrate how .width work.",
          .width = 40)

# Let's add a leader
cat_magic("Let's write a very long message to illustrate how `.width` work.",
          "And now we add `.leader`.", .width = 40, .leader = "#> ")

# newlines respect the introductory spaces
cat_magic("Here's a list:",
          "    + short item",
          "    + this is a very long item that likely overflows",
         .width = 30, .sep = "\n")
```

`get_interpolated_expr` *Lists the expressions used for interpolation in a* `string_magic` *call*

Description

Tool intended for development: use `get_interpolated_expr` to obtain the list of expressions which will be interpolated in a `string_magic()` call. The function `get_interpolated_vars` provides the variables instead.

Usage

```
get_interpolated_expr(x, parse = FALSE, delim = c("{", "}"))

get_interpolated_vars(x, delim = c("{", "}"))
```

Arguments

`x`	A character scalar for which the variables will be recovered. For example `x = "Hi {person}"` will return `"person"` (the variable that will be interpolated).
`parse`	Logical scalar, default is `FALSE`. If `TRUE`, the result is a list of R expressions. If `FALSE` (default), the result is a character vector of expressions.
`delim`	Character vector of length 1 or 2. Default is `c("{", "}")`. Defines the opening and the closing delimiters for interpolation. If of length 1, it must be of the form: 1) the opening delimiter, 2) a single space, 3) the closing delimiter. Ex: `".[ ]"` is equivalent to `c(".[", "]")`. The default value is equivalent to `"{ }"`. [ ]: R:%20 [", "]: R:%22,%20%22

Details

Note that this function captures even deeply nested interpolations.

Value

If the argument `parse = FALSE`, the default, then this function returns a character vector containing all the expressions that will be interpolated. This vector can be empty if there is no interpolation.

If the argument `parse = TRUE`, then a list is returned, containing the R expressions.

The function `get_interpolated_vars` always return a character vector.

Functions

- `get_interpolated_vars()`: Obtain the variables used in `string_magic()` interpolations

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

`parse_regex_pattern` `stringmagic`*'s regular expression parser*

Description

Parse regular expression with custom flags and obtain the final pattern to be parsed as well as the vector of flags

Usage

```
parse_regex_pattern(
  pattern,
  authorized_flags,
  parse_flags = TRUE,
  parse_logical = TRUE,
  envir = parent.frame()
)
```

Arguments

`pattern`	Character scalar, the regular expression pattern to parse.
`authorized_flags`	Character vector representing the flags to be parsed. Use the empty string if no flags are allowed.
`parse_flags`	Logical scalar, default is `TRUE`. Whether to parse the optional regex flags.
`parse_logical`	Logical scalar, default is `TRUE`. Whether to parse logical regex operations, similarly to `string_get()`.
`envir`	An environment, default is `parent.frame()`. Only used if the flag `magic` is present, it is used to find the variables to be interpolated.

Details

This is an internal tool that is exposed in order to facilitate checking what's going on.

There is no error handling.

Value

This function always returns a list of 4 elements:

- `flags`: the character vector of flags. If no flags were found, this is the empty string.
- `patterns`: the vector of regex patterns.
- `is_or`: logical vector of the same length as `patterns`. Indicates for each pattern if it should be attached to the previous patterns with a logical OR (`FALSE` means a logical `AND`).
- `is_not`: logical vector of the same length as `patterns`. Indicates for each pattern if it should be negated.

Author(s)

Laurent Berge

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
parse_regex_pattern("f/hello", c("fixed", "ignore"))

x = "john"
parse_regex_pattern("fm/{x} | Doe", c("fixed", "ignore", "magic"))
```

`paste_conditional` *Paste a string vector conditionally*

Description

Easily reconstruct a string vector that has been split with `string_split2df()`.

Usage

```
paste_conditional(x, id, sep = " ", names = TRUE, sort = TRUE)
```

Arguments

`x`	A character vector or a formula. If a vector: it represents the values to be pasted together. If a formula, it must be of the form `my_string ~ id1 + id2` with on the left the character vector and on the right the (possibly many) identifiers. If a formula, the argument `id` must be a data frame.
`id`	A vector of identifiers, a list of identifiers (can be a data frame), or a `data.frame`. The identifiers can be a vector or a list of vectors. They represent which elements of `x` are to be pasted together. When `x` is a formula, then `id` must be a data.frame containing the variables in the formula.
`sep`	A character scalar, default is `" "`. The value used to paste together the elements of `x`.
`names`	Logical scalar, default is `TRUE`. Whether to add, as names, the values of the identifiers.
`sort`	Logical scalar, default is `TRUE`. Whether to sort the results according to the identifiers.

Value

Returns a character vector. If the argument `names` is `TRUE` (default), the vector will have names equal to the values of the identifiers.

Examples

```
#
# let's paste together the letters of the alphabet

# first we create the identifier
id = rep(1:2, each = 13)
setNames(id, letters)

# now we conditionally paste together the letters
paste_conditional(letters, id, "")

#
# using a formula

# we create a small data set based on mtcars
base_cars = within(mtcars, carname <- row.names(mtcars))
base_cars = head(base_cars, 10)

# we use two identifiers
paste_conditional(carname ~ gear + carb, base_cars, sep = ", ")
```

`string_clean_alias` *Cleans a character vector from multiple patterns*

Description

Recursively cleans a character vector from several patterns. Quickly handle the tedious task of data cleaning by taking advantage of the syntax. You can also apply all sorts of cleaning operations by summoning `string_ops()` operations.

Usage

```
string_clean_alias(
  replacement = "",
  pipe = " => ",
  split = ",[ \n\t]+",
  ignore.case = FALSE,
  fixed = FALSE,
  word = FALSE,
  total = FALSE,
  single = FALSE,
  namespace = NULL
)

string_clean(
  x,
  ...,
  replacement = "",
  pipe = " => ",
  split = ",[ \n\t]+",
  ignore.case = FALSE,
  fixed = FALSE,
  word = FALSE,
  total = FALSE,
  single = FALSE,
  envir = parent.frame(),
  namespace = NULL
)

string_replace(
  x,
  pattern,
  replacement = "",
  pipe = " => ",
  ignore.case = FALSE,
  fixed = FALSE,
  word = FALSE,
  total = FALSE,
  single = FALSE,
  envir = parent.frame()
)

stclean(
  x,
  ...,
  replacement = "",
  pipe = " => ",
  split = ",[ \n\t]+",
  ignore.case = FALSE,
  fixed = FALSE,
  word = FALSE,
  total = FALSE,
  single = FALSE,
  envir = parent.frame(),
  namespace = NULL
)

streplace(
  x,
  pattern,
  replacement = "",
  pipe = " => ",
  ignore.case = FALSE,
  fixed = FALSE,
  word = FALSE,
  total = FALSE,
  single = FALSE,
  envir = parent.frame()
)
```

Arguments

`replacement`	Character scalar, default is the empty string. It represents the default value by which the patterns found in the character strings will be replaced. For example `string_clean(x, "e", replacement = "a")` turn all letters "e" in `x` into "a".
`pipe`	Character scalar, default is `" => "`. If thevalue of `pipe` is found in a pattern, then the string is split w.r.t. the pipe and anything after the pipe becomes the replacement. For example in `string_clean(x, "e => a")` the default pipe is found in "e => a", so the pattern "e" will be replaced with "a". In other terms, this is equivalent to `string_clean(x, "e", replacement = "a")`. Example changing the pipe: you can obtain the previous result with `string_clean(x, "e|>a", pipe = "|>")`.
`split`	Character scalar, default is `",[ \t\n]+"` (which means a comma followed with spaces and/or new lines). By default the patterns to be replaced are comma separated, that is the pattern is split w.r.t. the argument `split` and a replacement

	is done for each sub-pattern. Use `NULL` or the empty string to disable pattern separation. For example: let's look at `string_clean(x, "w/one, two => three")`. First the flag "word" is extracted from the pattern (see arg. `...`) as well as the replacement (see arg. `pipe`), leading to "one, two" the pattern to be replaced. Then the pattern is split w.r.t. `split`, leading to two patterns "one" and "two". Hence the words (thanks to the flag "w") "one" and "two" from the string `x` will be replaced with "three".
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`fixed`	Logical scalar, default is `FALSE`. Whether to trigger a fixed search instead of a regular expression search (default).
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then pattern = "The, mountain" will select strings containing either the word 'The' or the word 'mountain'.
`total`	Logical scalar, default is `FALSE`. If `TRUE`, then when a pattern is found in a string, the full string is replaced (instead of just the pattern). Note, *importantly*, that when `total = TRUE` you can use logical operators in the patterns. Example: `string_clean(x, "wi/ & two, three & !four => ", total = TRUE)`
`single`	Logical scalar, default is `FALSE`. Whether, in substitutions, to stop at the first match found. Ex: `string_clean("abc", "[[:alpha:]] => _", single = TRUE)` leads to `"_bc"`, while `string_clean("abc", "[[:alpha:]] => _")` leads to `"___"`.
`namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`. If so pass the name of your package in this argument so that your function can access the new `string_magic` operations defined within your package.
`x`	A character vector.
`...`	Character scalars representing patterns. A pattern is of the form "flags/pat1, pat2 => replacement". This means that patterns 'pat1' and 'pat2' will be replaced with the string 'replacement'. By default patterns are comma separated and the replacement comes after a ' => ' (see args `split` and `pipe` to change this). By default the replacement is the empty string (so "pat1, pat2" *removes* the patterns). Available regex flags are: 'word' (add word boundaries), 'ignore' (the case), 'fixed' (no regex), 'total', 'single' and 'magic'. The flag `total` leads to a *total replacement* of the string if the pattern is found. The flag 'magic' allows to interpolate variables within the pattern. Use flags with comma separation ("word, total/pat") or use only their initials ("wt/pat"). Starting with an '@' leads to operations in `string_ops()`. Ex: "@ascii, lower, ws" turns the string into ASCII, lowers the case and normalizes white spaces (see help of `string_ops()`).

`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.
`pattern`	A character scalar containing a regular expression pattern to be replaced. You can write the replacement directly in the string after a pipe: ' => ' (see arg. `pipe` to change this). By default the replacement is the empty string (so "pat1" *removes* the pattern). Available regex flags are: 'word' (add word boundaries), 'ignore' (the case), 'fixed' (no regex), 'total', 'single' and 'magic'. The flag `total` leads to a *total replacement* of the string if the pattern is found. The flag 'magic' allows to interpolate variables within the pattern. Use flags with comma separation ("word, total/pat") or use only their initials ("wt/pat").

Value

The main usage returns a character vector of the same length as the vector in input. Note, however, that since you can apply arbitrary `string_ops()` operations, the length and type of the final vector may depend on those (if they are used).

Functions

- `string_clean_alias()`: Create a `string_clean` alias with custom defaults
- `string_replace()`: Simplified version of `string_clean`
- `stclean()`: Alias to `string_clean`
- `streplace()`: Alias to `string_replace`

Regular expression flags specific to replacement

This function benefits from two specific regex flags: "total" and "single".

- "total" replaces the *complete string* if the pattern is found (remember that the default behavior is to replace just the pattern).
- "single" performs a single substitution for each string element and stops there. Only the first match of each string is replaced. Technically we use `base::sub()` internally instead of `base::gsub()`.

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (`/`). Example: `string_which(c("hello...", "world"), "fixed/.")` returns `1`. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion `string_which(c("hello...", "world"), ".")` returns `1:2`.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries (`"\\b"` in regex language) to the pattern. Further, the comma (`","`) becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: `string_clean("Am I ambushed?", "wi/am")` leads to " I ambushed?" thanks to the flags "ignore" and "word".
- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if `letters = "aiou"` then `string_clean("My great goose!", "magic/[{letters}] => e")` leads to `"My greet geese!"`

Author(s)

Laurent R. Berge

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
x = c("hello world  ", "it's 5 am....")

# we clean the o's and the points (we use 'fixed' to trigger fixed-search)
string_clean(x, "o", "f/.")
# equivalently
string_clean(x, "fixed/o, .")
# equivalently
string_clean(x, "o, .", fixed = TRUE)
# equivalently
string_clean(x, "o", ".", fixed = TRUE)

#
# chaining operations: example using cars
#

cars = row.names(mtcars)
new = string_clean(cars,
           # replace strings containing "Maz" with Mazda
           "total/Maz => Mazda",
           # replace the word 'Merc' with Mercedes
```

```
          "wi/merc => Mercedes",
          # replace strings containing "Merc" and a digit followed with an 'S'
          "t/Merc & \\dS => Mercedes S!",
          # put to lower case, remove isolated characters and normalize white spaces
          "@lower, ws.isolated")

cbind(cars, new)
```

string_extract   *Extracts a pattern from a character vector*

Description

Extracts the first, or several, patterns from a character vector.

Usage

```
string_extract(
  x,
  pattern,
  single = FALSE,
  simplify = TRUE,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  unlist = FALSE,
  envir = parent.frame()
)

stextract(
  x,
  pattern,
  single = FALSE,
  simplify = TRUE,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  unlist = FALSE,
  envir = parent.frame()
)
```

Arguments

x	A character vector.

`pattern`	A character scalar. It represents the pattern to be extracted from `x`. By default this is a regular expression. You can use flags in the pattern in the form `flag1, flag2/pattern`. Available flags are `ignore` (case), `fixed` (no regex), word (add word boundaries), single (select only the first element), and magic (add interpolation with `{}`). Example: if `"ignore/hello"` and `x = "Hello world` extracted text is `"Hello"`. Shortcut: use the first letters of the flags. Ex: "iw/one" will extract the word "one" (flags 'ignore' + 'word').
`single`	Logical scalar, default is `FALSE`. If `TRUE`, only the first pattern that is detected will be returned. Note that in that case, a character vector is returned of the same length as the vector in input.
`simplify`	Logical scalar, default is `TRUE`. If `TRUE`, then when the vector input `x` is of length 1, a character vector is returned instead of a list.
`fixed`	Logical scalar, default is `FALSE`. Whether to trigger a fixed search instead of a regular expression search (default).
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then pattern = "The, mountain" will select strings containing either the word 'The' or the word 'mountain'.
`unlist`	Logical scalar, default is `FALSE`. If `TRUE`, the function `unlist` is applied to the resulting list, leading to a character vector in output (instead of a list).
`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.

Value

The object returned by this functions can be a list or a character vector.

If `single = TRUE`, a character vector is returned, containing the value of the first match. If no match is found, an empty string is returned.

If `single = FALSE` (the default) and `simplify = TRUE` (default), the object returned is:

- a character vector if `x`, the vector in input, is of length 1: the character vector contains all the matches and is of length 0 if no match is found.
- a list of the same length as `x`. The ith element of the list is a character vector of the matches for the ith element of `x`.

If `single = FALSE` (default) and `simplify = FALSE`, the object returned is always a list.

Functions

- `stextract()`: Alias to `string_extract`

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (/). Example: `string_which(c("hello...", "world"), "fixed/.")` returns `1`. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion `string_which(c("hello...", "world"), ".")` returns `1:2`.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries (`"\\b"` in regex language) to the pattern. Further, the comma (`","`) becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: `string_clean("Am I ambushed?", "wi/am")` leads to " I ambushed?" thanks to the flags "ignore" and "word".
- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if `letters = "aiou"` then `string_clean("My great goose!", "magic/[{letters}] => e")` leads to `"My greet geese!"`

Examples

```
cars = head(row.names(mtcars))

# Let's extract the first word:
string_extract(cars, "\\w+", single = TRUE)

# same using flags
string_extract(cars, "s/\\w+")

# extract all words composed on only letters
# NOTE: we use the flag word (`w/`)
string_extract(cars, "w/[[:alpha:]]+")

# version without flag:
string_extract(cars, "\\b[[:alpha:]]+\\b")

# If a vector of length 1 => a vector is returned
greet = "Hi Tom, how's Mary doing?"
string_extract(greet, "w/[[:upper:]]\\w+")

# version with simplify = FALSE => a list is returned
string_extract(greet, "w/[[:upper:]]\\w+", simplify = FALSE)
```

`string_fill` *Fills a character string up to a size*

Description

Fills a character string up to a size and handles multibyte encodings (differently from sprintf).

Usage

```
string_fill(
  x = "",
  n = NULL,
  symbol = " ",
  right = FALSE,
  center = FALSE,
  center.right = TRUE,
  na = "NA"
)
```

Arguments

`x`	A character vector.
`n`	Integer scalar, possibly equal to `NULL` (default). The size up to which the character vector will be filled. If `NULL` (default), it is set to the largest width in the character vector `x`. To handle how the character is filled, see the arguments `symbol`, `right` and `center`.
`symbol`	Character scalar of length 1, default is a space (" "). It is the symbol with which the string will be filled.
`right`	Logical scalar, default is `FALSE`. If `TRUE`, then the filling of the string is done from the left, leading to right-alignment.
`center`	Logical scalar, default is `FALSE`. If `TRUE`, then the filling of the string will be balanced so as to center the strings.
`center.right`	Logical scalar, default is `TRUE`. Only used when `center = TRUE`, ignored otherwise. If `TRUE`, then when the width is odd and the number of characters of the string is even (or vice versa), the text is centered with one character on the right. If `FALSE`, this is one character on the left.
`na`	Character scalar or `NA`. Default is "NA" (a character string!). What happens to NAs: by default they are replaced by the character string "NA".

Details

If you use character filling of the form `sprintf("% 20s", x)` with `` x``containing multibyte characters, you may be s``
uses only base R functions to compensate this. It is slightly slower but, in general, safer.

It also looks a bit like `base::format()`, but slightly different (and a bit faster, but more restrictive).

Value

This functions returns a character vector of the same lenght as the vector in input.

Author(s)

Laurent R. Berge

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
x = c("apple", "pineapple")

# simple fill with blank
cat(paste0(string_fill(x), ":", c(3, 7), "€"), sep = "\n")

# center fill
cat(paste0(string_fill(x, center = TRUE), ":", c(3, 7), "€"), sep = "\n")

# changing the length of the fill and the symbol used for filling
cat(paste0(string_fill(x), ":",
           string_fill(c(3, 7), 3, "0", right = TRUE), "€"), sep = "\n")

# na behavior: default/NA/other
x = c("hello", NA)
string_fill(x)
string_fill(x, na = NA)
string_fill(x, na = "(missing)")
```

`string_get` *Gets elements of a character vector*

Description

Convenient way to get elements from a character vector.

Usage

```
string_get(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  seq = FALSE,
  seq.unik = FALSE,
  pattern = NULL,
  envir = parent.frame()
)

stget(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  seq = FALSE,
  seq.unik = FALSE,
  pattern = NULL,
  envir = parent.frame()
)
```

Arguments

`x`	A character vector.
`...`	Character scalars representing the patterns to be found. By default they are (perl) regular-expressions. Use ' & ' or ' | ' to chain patterns and combine their result logically (ex: `'[[:alpha:]] & \\d'` gets strings containing both letters and numbers). You can negate by adding a ! first (ex: `"!sepal$"` will return `TRUE` for strings that do not end with `"sepal"`). Add flags with the syntax 'flag1, flag2/pattern'. Available flags are: 'fixed', 'ignore', 'word' and 'magic'. Ex: "ignore/sepal" would get "Sepal.Length" (wouldn't be the case w/t 'ignore'). Shortcut: use the first letters of the flags. Ex: "if/dt[" would get `"DT[i = 5]"` (flags 'ignore' + 'fixed'). For 'word', it adds word boundaries to the pattern. The `magic` flag first interpolates values directly into the pattern with "".
`fixed`	Logical scalar, default is `FALSE`. Whether to trigger a fixed search instead of a regular expression search (default).
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then

	pattern = "The, mountain" will select strings containing either the word ’The’ or the word ’mountain’.
`or`	Logical, default is `FALSE`. In the presence of two or more patterns, whether to combine them with a logical "or" (the default is to combine them with a logical "and").
`seq`	Logical, default is `FALSE`. The argument `pattern` accepts a vector of patterns which are combined with an `and` by default. If `seq = TRUE`, then it is like if `string_get` was called sequentially with its results stacked. See examples.
`seq.unik`	Logical, default is `FALSE`. The argument `...` (or the argument `pattern`) accepts a vector of patterns which are combined with an `and` by default. If `seq.unik = TRUE`, then `string_get` is called sequentially with its results stacked, and `unique()` is applied in the end. See examples.
`pattern`	(If provided, elements of `...` are ignored.) A character vector representing the patterns to be found. By default a (perl) regular-expression search is triggered. Use ’ & ’ or ’ | ’ to chain patterns and combine their result logically (ex: `'[[:alpha:]] & \\d'` gets strings containing both letters and numbers). You can negate by adding a ! first (ex: `"!sepal$"` will return `TRUE` for strings that do not end with `"sepal"`). Add flags with the syntax ’flag1, flag2/pattern’. Available flags are: ’fixed’, ’ignore’, ’word’ and ’magic’. Ex: "ignore/sepal" would get "Sepal.Length" (wouldn’t be the case w/t ’ignore’). Shortcut: use the first letters of the flags. Ex: "if/dt[" would get `"DT[i = 5]"` (flags ’ignore’ + ’fixed’). For ’word’, it adds word boundaries to the pattern. The `magic` flag first interpolates values directly into the pattern with "".
`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.

Details

This function is a wrapper to `string_is()`.

Value

It always return a character vector.

Functions

- `stget()`: Alias to `string_get`

Caching

In an exploratory stage, it can be useful to quicky get values from a vector with the least hassle as possible. Hence `string_get` implements caching, so that users do not need to repeat the value of the argument `x` in successive function calls, and can concentrate only on the selection patterns.

Caching is a feature only available when the user calls `string_get` from the global environment. If that feature were available in regular code, it would be too dangerous, likely leading to hard to debug bugs. Hence caching is disabled when used within code (i.e. inside a function or inside an automated script), and function calls without the main argument will lead to errors in such scripts.

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (`/`). Example: `string_which(c("hello...", "world"), "fixed/.")` returns `1`. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion `string_which(c("hello...", "world"), ".")` returns `1:2`.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries (`"\\b"` in regex language) to the pattern. Further, the comma (`","`) becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: `string_clean("Am I ambushed?", "wi/am")` leads to " I ambushed?" thanks to the flags "ignore" and "word".
- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if `letters = "aiou"` then `string_clean("My great goose!", "magic/[{letters}] => e")` leads to `"My greet geese!"`

Author(s)

Laurent R. Berge

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
x = rownames(mtcars)

# find all Mazda cars
string_get(x, "Mazda")
```

```
# same with ignore case flag
string_get(x, "i/mazda")

# all cars containing a single digit (we use the 'word' flag)
string_get(x, "w/\\d")

# finds car names without numbers AND containing `u`
string_get(x, "!\\d", "u")
# equivalently
string_get(x, "!\\d & u")

# Stacks all Mazda and Volvo cars. Mazda first
string_get(x, "Mazda", "Volvo", seq = TRUE)

# Stacks all Mazda and Volvo cars. Volvo first
string_get(x, "Volvo", "Mazda", seq = TRUE)

# let's get the first word of each car name
car_first = string_ops(x, "extract.first")
# we select car brands ending with 'a', then ending with 'i'
string_get(car_first, "a$", "i$", seq = TRUE)
# seq.unik is similar to seq but applies unique()
string_get(car_first, "a$", "i$", seq.unik = TRUE)

#
# flags
#

# you can combine the flags
x = string_magic("/One, two, one... Two!, Microphone, check")
# regular
string_get(x, "one")
# ignore case
string_get(x, "i/one")
# + word boundaries
string_get(x, "iw/one")

# you can escape the meaning of ! with backslashes
string_get(x, "\\!")

#
# Caching
#

# Caching is enabled when the function is used interactively
# so you don't need to repeat the argument 'x'
# Mostly useful at an exploratory stage

if(interactive() && is.null(sys.calls())){

   # first run, the data is cached
   string_get(row.names(mtcars), "i/vol")
```

```
    # now you don't need to specify the data
    string_get("i/^m & 4")
}
```

`string_is` *Detects whether a pattern is in a character string*

Description

Function that detects if one or more patterns are in a string. The patterns can be chained, by default this is a regex search but special flags be triggered with a specific syntax, supports negation.

Usage

```
string_is(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame(),
  last = NULL
)

string_any(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
)

string_all(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
)

string_which(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
)

stis(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame(),
  last = NULL
)

stany(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
)

stall(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
```

```
)

stwhich(
  x,
  ...,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  or = FALSE,
  pattern = NULL,
  envir = parent.frame()
)
```

Arguments

`x`	A character vector.
`...`	Character scalars representing the patterns to be found. By default they are (perl) regular-expressions. Use ' & ' or ' | ' to chain patterns and combine their result logically (ex: `'[[:alpha:]] & \\d'` gets strings containing both letters and numbers). You can negate by adding a ! first (ex: `"!sepal$"` will return `TRUE` for strings that do not end with `"sepal"`). Add flags with the syntax 'flag1, flag2/pattern'. Available flags are: 'fixed', 'ignore', 'word' and 'magic'. Ex: "ignore/sepal" would get "Sepal.Length" (wouldn't be the case w/t 'ignore'). Shortcut: use the first letters of the flags. Ex: "if/dt[" would get `"DT[i = 5]"` (flags 'ignore' + 'fixed'). For 'word', it adds word boundaries to the pattern. The `magic` flag first interpolates values directly into the pattern with "".
`fixed`	Logical scalar, default is `FALSE`. Whether to trigger a fixed search instead of a regular expression search (default).
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then pattern = "The, mountain" will select strings containing either the word 'The' or the word 'mountain'.
`or`	Logical, default is `FALSE`. In the presence of two or more patterns, whether to combine them with a logical "or" (the default is to combine them with a logical "and").
`pattern`	(If provided, elements of `...` are ignored.) A character vector representing the patterns to be found. By default a (perl) regular-expression search is triggered. Use ' & ' or ' | ' to chain patterns and combine their result logically (ex: `'[[:alpha:]] & \\d'` gets strings containing both letters and numbers). You can negate by adding a ! first (ex: `"!sepal$"` will return `TRUE` for strings that do not end with `"sepal"`). Add flags with the syntax 'flag1, flag2/pattern'. Available flags are: 'fixed', 'ignore', 'word' and 'magic'. Ex: "ignore/sepal" would get "Sepal.Length" (wouldn't be the case w/t 'ignore'). Shortcut: use the first letters of the flags. Ex: "if/dt[" would get `"DT[i = 5]"` (flags 'ignore' + 'fixed'). For 'word', it adds word boundaries to the pattern. The `magic` flag first interpolates values directly into the pattern with "".

`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.
`last`	A function or `NULL` (default). If a function, it will be applied to the vector just before returning it.

Details

The internal function used to find the patterns is `base::grepl()` with `perl = TRUE`.

Value

It returns a logical vector of the same length as `x`.

The function `string_which` returns a numeric vector.

Functions

- `string_any()`: Detects if at least one element of a vector matches a regex pattern
- `string_all()`: Detects if all elements of a vector match a regex pattern
- `string_which()`: Returns the indexes of the values in which a pattern is detected
- `stis()`: Alias to `string_is`
- `stany()`: Alias to `string_any`
- `stall()`: Alias to `string_all`
- `stwhich()`: Alias to `string_which`

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (`/`). Example: `string_which(c("hello...", "world"), "fixed/.")` returns `1`. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion `string_which(c("hello...", "world"), ".")` returns `1:2`.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries (`"\\b"` in regex language) to the pattern. Further, the comma (`","`) becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: `string_clean("Am I ambushed?", "wi/am")` leads to " I ambushed?" thanks to the flags "ignore" and "word".

- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if `letters = "aiou"` then `string_clean("My great goose!", "magic/[{letters}] => e")` leads to `"My greet geese!"`

Author(s)

Laurent R. Berge

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
# NOTA: using `string_get` instead of `string_is` may lead to a faster understanding
#       of the examples

x = string_vec("One, two, one... two, microphone, check")

# default is regular expression search
# => 3 character items
string_is(x, "^...$")

# to trigger fixed search use the flag 'fixed'
string_is(x, "fixed/...")
# you can just use the first letter
string_is(x, "f/...")

# to negate, use '!' as the first element of the pattern
string_is(x, "f/!...")

# you can combine several patterns with " & " or " | "
string_is(x, "one & c")
string_is(x, "one | c")

#
# word: adds word boundaries
#

# compare
string_is(x, "one")
# with
string_is(x, "w/one")
```

```
# words can be chained with commas (it is like an OR logical operation)
string_is(x, "w/one, two")
# compare with
string_is(x, "w/one & two")
# remember that you can still negate
string_is(x, "w/one & !two")

# you can combine the flags
# compare
string_is(x, "w/one")
# with
string_is(x, "wi/one")

#
# the `magic` flag
#

p = "one"
string_is(x, "m/{p}")
# Explanation:
# - "p" is interpolated into "one"
# - we get the equivalent: string_is(x, "one")


#
# string_which
#

# it works exactly the same way as string_is
# Which are the items containing an 'e' and an 'o'?
string_which(x, "e", "o")
# equivalently
string_which(x, "e & o")
```

`string_magic` *String interpolation with operation chaining*

Description

This is firstly a string interpolation tool. On top of this it can apply, and chain, over 50 basic string operations to the interpolated variables. Advanced support for pluralization.

Usage

```
string_magic(
  ...,
  .envir = parent.frame(),
  .data = list(),
  .sep = "",
  .vectorize = FALSE,
  .delim = c("{", "}"),
  .last = NULL,
  .post = NULL,
  .nest = FALSE,
  .collapse = NULL,
  .invisible = FALSE,
  .default = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .class = NULL,
  .help = NULL,
  .namespace = NULL
)

.string_magic(
  ...,
  .envir = parent.frame(),
  .data = list(),
  .sep = "",
  .vectorize = FALSE,
  .delim = c("{", "}"),
  .collapse = NULL,
  .last = NULL,
  .nest = FALSE,
  .trigger = TRUE,
  .namespace = NULL
)

sma(
  ...,
  .envir = parent.frame(),
  .data = list(),
  .sep = "",
  .vectorize = FALSE,
  .delim = c("{", "}"),
  .last = NULL,
  .post = NULL,
  .nest = FALSE,
  .collapse = NULL,
  .invisible = FALSE,
  .default = NULL,
  .trigger = TRUE,
  .check = TRUE,
  .class = NULL,
  .help = NULL,
  .namespace = NULL
)
```

Arguments

`...`	Character scalars that will be collapsed with the argument `sep`. Note that named arguments are used for substitution. To interpolate, you can use `"{x}"` within each character string to insert the value of `x` in the string. You can add string operations in each `"{}"` instance with the syntax `"'arg'op ? x"` (resp. `"'arg'op ! x"`) to apply the operation `'op'` with the argument `'arg'` to `x` (resp. the verbatim of `x`). Otherwise, what to say? Ah, nesting is enabled, and since there's over 50 operators, it's a bit complicated to sort you out in this small space. Use the argument `.help = "keyword"` (or `.help = TRUE`) to obtain a selective help from the main documentation. Note that in interpolations you have access to the special variables: `.now` and `.date` to get the current time; and the special function `.now("format")` to format the time. Ex: `.now('%Y-%m %H:%m')`.
`.envir`	An environment used to evaluate the variables in `"{}"`. By default the variables are evaluated using the environment from where the function is called or using the named arguments passed to the function.
`.data`	A list used to evaluate the variables in `"{}"`. Default is the empty list. By default the variables are evaluated using the environment from where the function is called or using the named arguments passed to the function.
`.sep`	Character scalar, default is the empty string `""`. It is used to collapse all the elements in `...` before applying any operation.
`.vectorize`	Logical scalar, default is `FALSE`. If `TRUE`, Further, elements in `...` are NOT collapsed together, but instead vectorised.
`.delim`	Character vector of length 1 or 2. Default is `c("{", "}")`. Defines the opening and the closing delimiters for interpolation. If of length 1, it must be of the form: 1) the opening delimiter, 2) a single space, 3) the closing delimiter. Ex: `".[ ]"` is equivalent to `c(".[", "]")`. The default value is equivalent to `"{ }"`. [ ]: R:%20 [", "]: R:%22,%20%22
`.last`	Character scalar, a function, or `NULL` (default). If provided and character: it must be an `string_magic` chain of operations of the form `"'arg1'op1, op2, etc"`. All these operations are applied just before returning the vector. If a function, it will be applied to the resulting vector.
`.post`	Function or `NULL` (default). If not `NULL`, this function will be applied after all the processing, just before returning the object. This function can have extra arguments which will be caught directly in the `...` argument of `string_magic`. For example if `.post = head`, you can directly pass the argument `n = 3` to `string_magic`'s arguments.
`.nest`	Logical, default is `FALSE`. If `TRUE`, it will nest the original string within interpolation delimiters, so that you can apply operations directly on the string. Example: `string_magic("upper ! hello")` returns "upper ! hello", while `string_magic("upper ! hello", .nest = TRUE)` returns `"HELLO"`.

`.collapse`	Character scalar, default is `NULL`. If provided, the character vector that should be returned is collapsed with the value of this argument. This leads to return a string of length 1.
`.invisible`	Logical scalar, default is `FALSE`. Whether the object returned should be invisible (i.e. not printed on the console).
`.default`	Character scalar or `NULL` (default). If provided, it must be a sequence of `string_magic` operations. It will be applied as a default to any interpolation. Ex: if `x = 1:2`, then `string_magic("x = {x}", .default = "enum")` leads to "x = 1 and 2", and is equivalent to `string_magic("x = {enum?x}")`. Note that this default operations does not apply to nested expressions. That is `string_magic("{!x{1:2}}", .default = "enum")` leads to `c("x1", "x2")` and NOT `"x1 and 2"`.
`.trigger`	Logical, default is `TRUE`. If `FALSE`, this function is not run. Can be useful in debugging situations where we want conditional evaluations.
`.check`	Logical scalar, default is `TRUE`. Whether to enable error-handling (i.e. human readable error messages). Without error-handling you can save something of the order of 40us. Useful only in long loops.
`.class`	Character vector representing the class to give to the object returned. By default it is `NULL`. Note that the class `string_magic` has a specific `print` method, usually nicer for small vectors (it `base::cat()`s the elements).
`.help`	Character scalar or `TRUE`, default is `NULL`. This argument is used to generate a dynamic help on the console. If `TRUE`, the user can select which topic to read from the main documentation, with the possibility to search for keywords and navigate the help pages. If a character scalar, then a regex search is perfomed on the main documentation and any section containining a match is displayed. The user can easily navigate across matches.
`.namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`.

Details

There are over 50 basic string operations, it supports pluralization, string operations can be nested, operations can be applied group-wise or conditionally and operators have sensible defaults. You can also declare your own operations with `string_magic_register_fun()` or `string_magic_register_ops()`. They will be seamlessly integrated to `string_magic`.

The function `.string_magic` (prefixed with a dot) is a leaner version of the function `string_magic`. It does the same operations but with the following differences:

- there is no error handling: meaning that the error messages, if any, will be poor and hard to understand
- default options are not applied: hence the user must always explicitly provide the arguments

This leads to a faster processing time (of about 50 microseconds) at the cost of user experience.

If you want to change the default values of `string_magic` (like changing the delimiter), use the function `string_magic_alias()`.

Use the argument `.help` to which you can pass keywords or regular expressions and fecth select pieces from the main documentation.

Value

It returns a character vector whose length depends on the elements and operations in the interpolations.

Functions

- `string_magic()`: String interpolation with operation chaining
- `.string_magic()`: A simpler version of `string_magic` without any error handling to save a few micro seconds
- `sma()`: Alias to `string_magic`

Interpolation and string operations

Principle:

To interpolate a variable, say `x`, simply use `{x}`. For example `x = "world"; string_magic("hello {x}")` leads to "hello world".

To any interpolation you can add operations. Taking the previous example, say we want to display "hello W O R L D". This means upper casing all letters of the interpolated variable and adding a space between each of them. Do you think we can do that? Of course yes: `string_magic("hello {upper, ''s, c ? x}")`. And that's it.

Now let's explain what happened. Within the `{}` *box*, we first write a set of operations, here "upper, ”s, c", then add "?" and finally write the variable to interpolate, "x". The operations (explained in more details below) are `upper`, upper-casing all letters, ”s: splitting with the empty string, 'c': concatenating with spaces the vector of string that was just split. The question mark means that the expression coming after it is to be evaluated (this is opposed to the exclamation mark presented next).

The syntax is always the same: `{operations ? expression}`, where the operations section is a *comma separated* list of operations. These operations are of the form `'arg'op`, with `arg` the argument to the operator code `op`. These operations are performed sequantially from left to right.

Some operations, like `upper`, accept options. You attach options to an operation with a dot followed by the option name. Formally: `op.option1.option2`, etc. Example: `x = "hi there. what's up? fine." ; string_magic`
Leads to: `He said: "Hi there. What's up? Fine."`.

Both operators and options are partially matched. So `string_magic("He said: {up.s, Q ? x}")` would also work.

Verbatim interpolation and nesting

Principle:

Instead of interpolating a variable, say `x`, with `{x}`, you can use an exclamation mark to trigger varbatim evaluation. For example `string_magic("hello {!x}")` would lead to "hello `x`". It's a bit disappointing, right? What's the point of doing that? Wait until the next two paragraphs.

Verbatim evaluation is a powerful way to apply operations to plain text. For example: `string_magic("hello {upper, ''s, c ! world}")` leads to "hello W O R L D".

(A note in passing. The spaces surrounding the exclamation mark are non necessary, but when one space is present on both sides of the `!`, then the verbatim expression only begins after it. Ex:

`"{upper! hi}"` leads to " HI" while `"{upper ! hi}"` leads to "HI" and `"{upper ! hi}"` leads to " HI".)

The second advantage of verbatim evaluations is *nesting*. Anything in a verbatim expression is evaluated with the function `string_magic`. This means that any *box* will be evaluated as previously described. Let's give an example. You want to write the expression of a polynomial of order n: a + bx + cx^2 + etc. You can do that with nesting. Assume we have `n = 2`.

Then `string_magic("poly({n}): {' + 'c ! {letters[1 + 0:n]}x^{0:n}}")` leads to "poly(2): ax^0 + bx^1 + cx^2".

How does it work? The verbatim expression (the one following the exclamation mark), here `"{letters[1 + 0:n]}x^{0:n}"`, is evaluated with `string_magic`. `string_magic("{letters[1 + 0:n]}x^{0:n}")` leads to the vector c("ax^0", "bx^1", "cx^2").

The operation `' + 'c` then concatenates (or collapses) that vector with ' + '. This value is then appended to the previous string.

We could refine by adding a cleaning operation in which we replace "x^0" and "^1" by the empty string. Let's do it:

`string_magic("poly({n}): {' + 'c, 'x\\^0|\\^1'r ! {letters[1 + 0:n]}x^{0:n}}")` leads to "poly(2): a + bx + cx^2", what we wanted.

You can try to write a function to express the polynomial as before: although it is a simple task, my guess is that it will require more typing.

Operations

General syntax:

As seen in the previous sections, within a *box* (i.e. `"{}"`), multiple operations can be performed. We can do so by stacking the operations codes and in a comma separated enumeration. Operations can have arguments, and operations can also have options. The general syntax, with argument and options, is:
`{'arg1'op1.optionA.optionB, arg2 op2.optionC, arg3op3, 51op4 ? x}`

The argument can appear in four forms: a) inside single or double quotes just before the operation name (`arg1` above), b) verbatim, separated with a space, just before the operation name (`arg2` above), c) inside bactick quotes the argument is evaluated from the environment (`arg3` above), or d) when the argument is an integer it can be juxtaposed to the opeation name (like in `op4` above).

The options are always dot separated and attached to the operation name, they are specific to each operation.

Both the operation name and the option names are partially matched.

Basic string operations

This section describes some of the most common string operations: extracting, replacing, collapsing, splitting, etc. These functions accept generic flags ("ignore", "fixed", "word") in their patterns (syntax: "flags/pattern"). Please see the dedicated section for more information on flags.

- s, split, S, Split: splits the string according to a pattern. The operations have different defaults: `' '` for `s` and 'split', and `',[ \t\n]*'` for `S` and 'Split' (i.e. comma separation). Ex.1: `string_magic("{S ! romeo, juliet}")` leads to the vector c("romeo", "juliet"). Ex.2:

string_magic("{'f/+'s, '-'c ! 5 + 2} = 3") leads to "5 - 2 = 3" (note the flag "fixed" in s's pattern).

- c, C: to concatenate multiple strings into a single one. The two operations are identical, only their default change. c: default is ' ', C: default is ', | and '. The syntax of the argument is 's1' or 's1|s2'. s1 is the string used to concatenate (think paste(x, collapse = s1)). In arguments of the form 's1|s2', s2 will be used to concatenate the last two elements. Ex.1: x = 1:4; string_magic("Et {' et 'c ? x}!") leads to "Et 1 et 2 et 3 et 4!". Ex.2: string_magic("Choose: {', | or 'c ? 2:4}?") leads to "Choose: 2, 3 or 4?".
- x, X: extracts patterns from a string. Both have the same default: '[[:alnum:]]+'. x extracts the first match while X extracts **all** the matches. Ex.1: x = c("6 feet under", "mahogany") ; string_magic("{'\\w
leads to the vector c("fee", "mah"). Ex2.: x = c("6 feet under", "mahogany") ; string_magic("{'\\w{3}'X ? x}
leads to the vector c("fee", "und", "mah", "oga").
- extract: extracts multiple patterns from a string, this is an alias to the operation X described above. Use the option "first" to extract only the first match for each string (behavior becomes like x). Ex: x = c("margo: 32, 1m75", "luke doe: 27, 1m71") ; string_magic("{'^\\w+'extract ? x} is {'\
leads to c("margo is 32", "luke is 27").
- r, R: replacement within a string. The two operations are identical and have no default. The syntax is 'old' or 'old => new' with 'old' the pattern to find and new the replacement. If new is missing, it is considered the empty string. This operation also accepts the flag "total" which instruct to replace the fulll string in case the pattern is found. Ex.1: string_magic("{'e'r ! Where is the letter e?}") leads to "Whr is th lttr ?". Ex.2: string_magic("{'(?<!\\b)e
=> a'R ! Where is the letter e?}") leads to "Whara is tha lattar e?". Ex.3: string_magic("{'t/e
=> here'r ! Where is the letter e?}") leads to "here".
- clean: replacement with a string. Similar to the operation r, except that here the comma is a pattern separator, see detailed explanations in string_clean(). Ex: string_magic("{'f/[,
]'clean ! x[a]}") leads to "xa".
- get: restricts the string only to values respecting a pattern. This operation has no default. Accepts the options "equal" and "in". By default it uses the same syntax as string_get() so that you can use regex flags and include logical operations with ' & ' and ' | ' to detect patterns. If the option "equal" is used, a simple string equality with the argument is tested (hence no flags are accepted). If the option "in" is used, the argument is first split with respect to commas and then set inclusion is tested. Example: x = row.names(mtcars) ; string_magic("Mercedes models: {'Merc & [[:a
leads to "Mercedes models: 240D, 280C, 450SE, 450SL and 450SLC".
- is: detects if a pattern is present in a string, returns a logical vector. This operation has no default. Accepts the options "equal" and "in". By default it uses the same syntax as string_is() so that you can use regex flags and include logical operations with ' & ' and ' | ' to detect patterns. If the option "equal" is used, a simple string equality with the argument is tested (hence no flags are accepted). If the option "in" is used, the argument is first split with respect to commas and then set inclusion is tested. Mostly useful as the final operation in a string_ops() call. Example: x = c("Mark", "Lucas") ; string_magic("Mark? {'i/mark'is, C ? x}") leads to "Mark? TRUE and FALSE".
- which: returns the index of string containing a specified pattern. With no default, can be applied to a logical vector directly. By default it uses the same syntax as string_which() so that you can use regex flags and include logical operations with ' & ' and ' | ' to detect patterns. If the option "equal" is used, a simple string equality with the argument is tested (hence no flags are accepted). If the option "in" is used, the argument is first split with respect to commas and

then set inclusion is tested. Mostly useful as the final operation in a `string_ops()` call. Ex.1: `x = c("Mark", "Lucas") ; string_magic("Mark is number {'i/mark'which ? x}.")` leads to "Mark is number 1.".

Operations changing the length or the order

- first: keeps only the first `n` elements. Example: `string_magic("First 3 numbers: {3 first, C ? mtcars$mpg}.")` leads to "First 3 numbers: 21, 21 and 22.8.". Negative numbers as argument remove the first `n` values. You can add a second argument in the form `'n1|n2'first` in which case the first `n1` and last `n2` values are kept; `n1` and `n2` must be positive numbers.
- K, Ko, KO: keeps only the first `n` elements; has more options than `first`. The syntax is `'n'K`, `'n|s'K`, `'n||s'K`. The values Ko and KO only accept the two first syntax (with `n` only). `n` provides the number of elements to keep. If `s` is provided and the number of elements are greater than `n`, then in 'n|s' the string `s` is added at the end, and if 'n||s' the string `s` replaces the nth element. The string `s` accepts specials values:
  - `:n:` or `:N:` which gives the total number of items in digits or letters (N)
  - `:rest:` or `:REST:` which gives the number of elements that have been truncated in digits or letters (REST) Ex: `string_magic("{'3|:rest: others'K ? 1:200}")` leads to the vector `c("1", "2", "3", "197 others")`.
  - The operator 'n'Ko is like `'n||:rest: others'K` and 'n'KO is like `'n||:REST: others'K`.
- last: keeps only the last `n` elements. Example: `string_magic("Last 3 numbers: {3 last, C ? mtcars$mpg}.")` leads to "Last 3 numbers: 19.7, 15 and 21.4.". Negative numbers as argument remove the last `n` values.
- sort: sorts the vector in increasing order. Accepts optional arguments and the option "num". Example: `x = c("sort", "me") ; string_magic("{sort, c ? x}")` leads to "me sort". If an argument is provided, it must be a regex pattern that will be applied to the vector using `string_clean()`. The sorting will be applied to the modified version of the vector and the original vector will be ordered according to this sorting. Ex: `x = c("Jon Snow", "Khal Drogo")`; `string_magic("{'.+ 'sort, C?x}")` leads to "Khal Drogo and Jon Snow". The option "num" sorts over a numeric version (with silent conversion) of the vector and reorders the original vector accordingly. Values which could not be converted are last. **Important note**: the sorting operation is applied before any character conversion. If previous operations were applied, it is likely that numeric data were transformed to character. Note the difference: `x = c(20, 100, 10); string_magic("{sort, ' + 'c ? x}")` leads to "10 + 20 + 100" while `string_magic("{n, sort, ' + 'c ? x}")` leads to "10 + 100 + 20" because the operation "n" first transformed the numeric vector into character.
- dsort: sorts the vector in decreasing order. It accepts an optional argument and the option "num". Example: `string_magic("5 = {dsort, ' + 'c ? 2:3}")` leads to "5 = 3 + 2". See the operation "sort" for a description of the argument and the option.
- rev: reverses the vector. Example: `string_magic("{rev, ''c ? 1:3}")` leads to "321".
- unik: makes the string vector unique. Example: `string_magic("Iris species: {unik, C ? iris$Species}.")` leads to "Iris species: setosa, versicolor and virginica.".
- table: computes the frequency of each element and attaches each element to its frequency. Accepts an argument which must be a character string representing a `string_magic` interpolation with the following variables: `x` (the element), `n` (its count) and `s` (its share). The default is `'{x} ({n ? n})'`. By default the resulting string vector is sorted by decreasing frequency.

You can change how the vector is sorted with five options: `sort` (sorts on the elements), `dsort` (decreasing sort), `fsort` (sorts on frequency), `dfsort` (decreasing sort on freq. – default), `nosort` (keeps the order of the first elements). Note that you can combine several sorts (to resolve the ties of elements with same frequencies). Example: `string_magic("Freq. of months: {'{x} ({n})'table, enum ? month.name[airquality$Month]}.")`

- each: repeats each element of the vector `n` times. Option "c" then collapses the full vector with the empty string as a separator. Ex.1: `string_magic("{/x, y}{2 each ? 1:2}")` leads to the vector `c("x1", "y1", "x2", "y2")`. Ex.2: `string_magic("Large number: 1{5 each.c ! 0}")` leads to "Large number: 100000".
- times: repeats the vector sequence `n` times. Option "c" then collapses the full vector with the empty string as a separator. Example: `string_magic("What{6 times.c ! ?}")` leads to "What??????".
- rm: removes elements from the vector. Options: "empty", "blank", "noalpha", "noalnum", "all". The *optional* argument represents the pattern used to detect strings to be deleted. Ex.1: `x = c("Luke", "Charles"); string_magic("{'i/lu'rm ? x}")` leads to "charles". By default it removes empty strings. Option "blank" removes strings containing only blank characters (spaces, tab, newline). Option "noalpha" removes strings not containing letters. Option "noalnum" removes strings not containing alpha numeric characters. Option "all" removes all strings (useful in conditions, see the dedicated section). If an argument is provided, only the options "empty" and "blank" are available. Ex.2: `x = c("I want to enter.", "Age?", "21"); string_magic("Nightclub conversation: {rm.noalpha, c ! - {x}}")` leads to "Nightclub conversation: - I want to enter. - Age?"
- nuke: removes all elements, equivalent to `rm.all` but possibly more explicit (not sure). Useful in conditions, see the dedicated section. Example: `x = c(5, 7, 453, 647); string_magic("Small numbers only: {`
  leads to "Small numbers only: 5 and 7";
- insert: inserts a new element to the vector. Options: "right" and "both". Option "right" adds the new element to the right. Option "both" inserts the new element on the two sides of the vector. Example: `string_magic("{'3'insert.right, ' + 'c ? 1:2}")` leads to "1 + 2 + 3".
- dp or `deparse`: Deparses an object and keeps only the first characters of the deparsed string. Accepts a number as argument. In that case only the first `n` characters are kept. Accepts option `long`: in that case all the lines of the deparsed object are first collapsed. Example: `fml = y ~ x1 + x2; string_magic("The estimated model is {dp ? fml}.")`

Formatting operations

- lower: lower cases the full string.
- upper: upper cases the full string. Options: "first" and "sentence". Option "first" upper cases only the first character. Option "sentence" upper cases the first letter after punctuation. Ex: `x = "hi. how are you? fine." ; string_magic("{upper.sentence ? x}")` leads to "Hi. How are you? Fine.".
- title: applies a title case to the string. Options: "force" and "ignore". Option "force" first puts everything to lowercase before applying the title case. Option "ignore" ignores a few small prepositions ("a", "the", "of", etc). Ex: `x = "bryan is in the KITCHEN" ; string_magic("{title.force.ignore ?`
  leads to "Bryan Is in the Kitchen".

- ws: normalizes whitespaces (WS). It trims the whitespaces on the edges and transforms any succession of whitespaces into a single one. Can also be used to further clean the string with its options. Options: "punct", "digit", "isolated". Option "punct" cleans the punctuation. Option "digit" cleans digits. Option "isolated" cleans isolated letters. WS normalization always come after any of these options. **Important note:** punctuation (or digits) are replaced with WS and **not** the empty string. This means that `string_magic("ws.punct ! Meg's car")` will become "Meg s car".
- trimws: trims the white spaces on both ends of the strings.
- q, Q, bq: to add quotes to the strings. q: single quotes, Q: double quotes, bq: back quotes. `x = c("Mark", "Pam"); string_magic("Hello {q, C ? x}!")` leads to "Hello 'Mark' and 'Pam'!".
- format, Format: applies the base R's function `base::format()` to the string. By default, the values are left aligned, *even numbers* (differently from `base::format()`'s behavior). The upper case command (`Format`) applies right alignment. Options: "0", "zero", "left", "right", "center". Options "0" or "zero" fills the blanks with 0s: useful to format numbers. Option "right" right aligns, and "center" centers the strings. Default is left alignment. Ex: `x = c(1, 12345); string_magic("left: {format.0, q, C ? x}, right: {Format, q, C ? x}")` leads to "left: '000001' and '12,345', right: ' 1' and '12,345'".
- %: applies `base::sprintf()` formatting. The syntax is 'arg'% with arg an sprintf formatting, or directly the sprint formatting, e.g. `% 5s`. Example: `string_magic("pi = {%.3f ? pi}")` leads to "pi = 3.142".
- stopwords: removes basic English stopwords (the snowball list is used). The stopwords are replaced with an empty space but the left and right WS are untouched. So WS normalization may be needed (see operation `ws`). `x = c("He is tall", "He isn't young"); string_magic("Is he {stop, ws, C ? x}` leads to "Is he tall and young?".
- ascii: turns all letters into ASCII with transliteration. Failed translations are transformed into question marks. Options: "silent", "utf8". By default, if some conversion fails a warning is prompted. Option "silent" disables the warning in case of failed conversion. The conversion is done with `base::iconv()`, option "utf8" indicates that the source endocing is UTF-8, can be useful in some cases.
- n: formats integers by adding a comma to separate thousands. Options: "letter", "upper", "0", "zero". The option "letter" writes the number in letters (large numbers keep their numeric format). The option "upper" is like the option "letter" but uppercases the first letter. Options "0" or "zero" left pads numeric vectors with 0s. Ex.1: `x = 5; string_magic("He's {N ? x} years old.")` leads to "He's five years old.". Ex.2: `x = c(5, 12, 52123); string_magic("She owes {n.0, '$'paste, C ? x}.")` leads to "She owes $5, $12 and $52,123.".
- N: same as `n` but automatically adds the option "letter".
- nth: when applied to a number, these operators write them as a rank. Options: "letter", "upper", "compact". Ex.1: `n = c(3, 7); string_magic("They finished {nth, enum ? n}!")` leads to "They finished 3rd and 7th!". Option "letter" tries to write the numbers in letters, but note that it stops at 20. Option "upper" is the same as "letter" but uppercases the first letter. Option "compact" aggregates consecutive sequences in the form "start_n_th to end_n_th". Ex.2: `string_magic("They arrived {nth.compact ? 5:20}.")` leads to "They arrived 5th to 20th.". Nth: same as `nth`, but automatically adds the option "letter". Example: `n = c(3, 7); string_magic("They finished {Nth, enum ? n}!")` leads to "They finished third and seventh!".

- ntimes: write numbers in the form `n` times. Options: "letter", "upper". Option "letter" writes the number in letters (up to 100). Option "upper" does the same as "letter" and uppercases the first letter. Example: `string_magic("They lost {C ! {ntimes ? c(1, 12)} against {S!Real, Barcelona}}.")` leads to "They lost once against Real and 12 times against Barcelona.".
- Ntimes: same as `ntimes` but automatically adds the option "letter". Example: `x = 5; string_magic("This paper was` leads to "This paper was rejected five times...".
- firstchar, lastchar: to select the first/last characters of each element. Ex: `string_magic("{19 firstchar, 9 lastchar ! This is a very long sentence}")` leads to "very long". Negative numbers remove the first/last characters.
- k: to keep only the first n characters (like `firstchar` but with more options). The argument can be of the form `'n'k`, `'n|s'k` or `'n||s'k` with `n` a number and `s` a string. `n` provides the number of characters to keep. Optionnaly, only for strings whose length is greater than `n`, after truncation, the string `s` can be appended at the end. The difference between 'n|s' and 'n||s' is that in the second case the strings will always be of maximum size `n`, while in the first case they can be of length `n + nchar(s)`. Ex: `string_magic("{4k ! long sentence}")` leads to "long", `string_magic("{'4|..'k ! long sentence}")` leads to "long..", `string_magic("{'4||..'k ! long sentence}")` leads to "lo..".
- fill, align (alias), width (alias): fills the character strings up to a size in order to fit a given width. Options: "left", "right", "center". Accepts arguments of the form `'n'` or `'n|s'`, with `n` a number and `s` a symbol. Default is left-alignment of the strings. Option "right" right aligns and "center" centers the strings. When using `'n|s'`, the symbol `s` is used for the filling. By default if no argument is provided, the maximum size of the character string is used. See help for `string_fill()` for more information. Ex.1: `string_magic("Numbers: {'5|0'fill.right, C ? c(1, 55)}")` leads to "Numbers: 00001 and 00055".
- paste, append: pastes some character to all elements of the string. This operation has no default. Options: "left", "both", "right", "front", "back", "delete". By default, a string is pasted on the left. By default, it pastes on the left. Option "right" pastes on the right and "both" pastes on both sides. Option "front" only pastes on the first element while option "back" only pastes on the last element. Option "delete" first replaces all elements with the empty string. Example: `string_magic("6 = {'|'paste.both, ' + 'c ? -3:-1}")` leads to "6 = |-3| + |-2| + |-1|". The argument can be of the form `s` or `s1|s2`. If of the second form, this is equivalent to chaining two `paste` operations, once on the left and once on the right: `'s1'paste, 's2'paste.right`.
- join: joins lines ending with a double backslash. Ex: `x = "the sun \\\n is shining"; string_magic("{join ? x}")` leads to "the sun is shining".
- escape: adds backslashes in front of specific characters. Options `"nl"`, `"tab"`. Option `"nl"` escapes the newlines (`\n`), leading them to be displayed as `"\\\\n"`. Option `"tab"` does the same for tabs (`"\t"`). This is useful to make the value free of space formatters. The default behavior is to escape both newlines and tabs.

Other operations

- num: converts to numeric. Options: "warn", "soft", "rm", "clear". By default, the conversion is performed silently and elements that failed to convert are turned into NA. Option "warns" displays a warning if the conversion to numeric fails. Option "soft" does not convert if the conversion of at least one element fails. Option "rm" converts and removes the elements that could not be converted. Option "clear" turns failed conversions into the empty string, and hence lead to a character vector. Example: `x = c(5, "six"); string_magic("Compare {num, C, q ? x} with {num.rm, C, q` leads to "Compare '5 and NA' with '5'.", and `string_magic("Compare {num.soft, C, q ? x} with {clear, C, q ? x}.")` leads to "Compare '5 and six' with '5 and '.".

- enum: enumerates the elements. It creates a single string containing the comma separated list of elements. If there are more than 7 elements, only the first 6 are shown and the number of items left is written. For example `string_magic("enum ? 1:5")` leads to "1, 2, 3, 4, and 5". You can add the following options by appending the letter to enum after a dot:
  - q, Q, or bq: to quote the elements
  - or, nor: to finish with an 'or' (or 'nor') instead of an 'and'
  - comma: to finish the enumeration with ", " instead of ", and".
  - i, I, a, A, 1: to enumerate with this prefix, like in: i) one, and ii) two
  - a number: to tell the number of items to display Ex.1: `x = c("Marv", "Nancy"); string_magic("The main char` leads to "The main characters are Marv and Nancy.". Ex.2: `x = c("orange", "milk", "rice"); string_magic(` leads to "Shopping list: i) 'orange', ii) 'milk', and iii) 'rice'."
- len: gives the length of the vector. Options "letter", "upper", "num". Option "letter" writes the length in words (up to 100). Option "upper" is the same as letter but uppercases the first letter. By default, commas are added to separate thousands. Use uption "num" to preserve a regular numeric format. Example: `string_magic("Size = {len ? 1:5000}")` leads to "Size = 5,000".
- swidth: stands for screen width. Formats the string to fit a given width by cutting at word boundaries and adding newlines appropriately. Accepts arguments of the form `'n'` or `'n|s'`, with `n` a number and `s` a string. An argument of the form `'n|s'` will add `s` at the beginning of each line. Further, by default a trailing white space is added to `s`; to remove this behavior, add an underscore at the end of it. The argument `n` is either an integer giving the target character width (minimum is 15), or it can be a fraction expressing the target size as a fraction of the current screen. Finally it can be an expression that uses the variable `.sw` which will capture the value of the current screen width. Ex.1: `string_magic("{15 swidth ! this is a long sentence}")` leads to "this is a long\nsentence". Ex.2: `string_magic("{15 swidth.#> ! this is a long sentence}")` leads to "#> this is a long\n#> sentence".
- difftime: displays a formatted time difference. Option "silent" does not report a warning if the operation fails. It accepts either objects of class `POSIXt` or `difftime`. Example: `x = Sys.time() ; Sys.sleep(0.5) ; string_magic("Time: {difftime ? x}")` leads to something like "Time: 514ms".

Group-wise operations

In `string_magic`, the splitting operation `s` (or `S`) keeps a memory of the strings that were split. Use the tilde operator, of the form `~(op1, op2)`, to apply operations group-wise, to each of the split strings.

Better with an example. `x = c("Oreste, Hermione", "Hermione, Pyrrhus", "Pyrrhus, Andromaque") ; string_magic("Troubles ahead: {S, ~(' loves 'c), C ? x}.")` leads to "Troubles ahead: Oreste loves Hermione, Hermione loves Pyrrhus and Pyrrhus loves Andromaque.".

Almost all operations can be applied group-wise (although only operations changing the order or the length of the strings really matter).

Conditional operations

There are two operators to apply operations conditionally: `if` and `vif`, the latter standing for *verbatim if*.

The syntax of `if` is `if(cond ; ops_true ; ops_false)` with `cond` a condition (i.e. logical operation) on the value being interpolated, `ops_true` a comma-separated sequence of operations if the condition is `TRUE` and `ops_false` an *optional* a sequence of operations if the condition is `FALSE`.

Ex.1: Let's take a sentence, delete words of less than 4 characters, and trim words of 7+ characters. x = "Songe Cephise a cette nuit cruelle qui fut pour tout un peuple une nuit eternelle" `string_magic("{' 's, if(.nchar<=4 ; nuke ; '7|..'k), c ? x}")`. Let's break it down. First the sentence is split w.r.t. spaces, leading to a vector of words. Then we use the special variable `.nchar` in `if`'s condition to refer to the number of characters of the current vector (the words). The words with less than 4 characters are nuked (i.e. removed), and the other words are trimmed at 7 characters. Finally the modified vector of words is collapsed with the function `c`, leading to the result.

The condition `cond` accepts the following special values: `.` (the dot), `.nchar`, `.C`, `.len`, `.N`. The dot, `.`, refers to the current vector. `.nchar` represent the number of characters of the current vector (equivalent to `nchar(.)`). `.C` is an alias to `.nchar`. `.len` represent the length of the current vector (equivalent to `length(.)`). `.N` is an alias to `.len`.

If a condition leads to a result of length 1, then the operations are applied to the full string vector and not element-wise (as was the case in Ex.1). Contrary to element-wise conditions for which operations modifying the length of the vectors are forbidden (apart from nuking), such operations are fine in full-string conditions.

Ex.2: `x = string_magic("x{1:10}")`; `string_magic("y = {if(.N>4 ; 3 first, '...'insert.right), ' + 'c ? x}")` leads to "y = x1 + x2 + x3 + ...". the same opration applied to `x = string_magic("x{1:4}")` leads to "y = x1 + x2 + x3 + x4".

For `vif`, the syntax is `vif(cond ; verb_true ; verb_false)` with `verb_true` a verbatim value with which the vector will be replaced if the condition is `TRUE`. This is similar for `verb_false`. The condition works as in `if`.

Ex.3: `x = c(1, 25, 12, 6) ; string_magic("Values: {vif(.<10 ; <10), C ? x}")` leads to "Values: <10, 25, 12 and <10". As we can see values lower than 10 are replaced with "<10" while other values are not modified.

Ex.4: `x = string_magic("x{1:10}")`; `string_magic("y = {vif(.N>4 ; {S!{x[1]}, ..., {last?x}}), ' + 'c ? x}")` leads to "y = x1 + ... + x10". Let's break it down. If the length of the vector is greater than 4 (here it's 10), then the full string is replaced with `"{S!{x[1]}, ..., {last?x}}"`. Interpolation applies to such string. Hence the split operation `S` breaks the string w.r.t. the commas (default behavior), leading to the vector `c("{x[1]}", "...", "{last?x}")`. Since the string contains curly brackets, interpolation is applied again. This leads to the vector `c("x1", "...", "x10")`. Finally, this vector is collapsed with ' + ' leading to the final string. Note that there are many ways to get to the same result. Here is another example: `string_magic("y = {vif(.N>4 ; {x[1]} + ... + {last?x} ; {' + 'c ? x}) ? x}")`.

The `vif` condition allows the use of '.' to refer to the current value in `verb_true` and `verb_false`, as illustrated by the last example:

Ex.5: `string_magic("{4 last, vif(. %% 2 ; x{.} ; y{rev?.}), C ? 1:11}")` leads to "y10, x9, y8 and x11".

Special interpolation

if-else:

Using an ampersand ("&") as the first character of an interpolation leads to an *if-else* operation. Using two ampersands ("&&") leads to a slightly different operation described at the end of this section.

The syntax is as follows: `{&cond ; verb_true ; verb_false}` with `cond` a condition (i.e. logical operation) on the value being interpolated, `verb_true` a verbatim value with which the vector will be replaced if the condition is `TRUE` and `verb_false` an *optional* verbatim value with which the vector will be replaced if the condition is `FALSE`. If not provided, `verb_false` is considered to be the empty string unless the operator is the double ampersand described at the end of this section.

Note that in `cond`, you can use the function `len`, an alias to `length`.

Ex.1: `x = 1:5; string_magic("x is {&len(x)<10 ; short ; {`log10(.N)-1`times, ''c ! very }long}")` leads to "x is short". With `x = 1:50`, it leads to "x is long", and to "x is very very long" if `x = 1:5000`.

If a condition leads to a result of length 1, the full string is replaced by the verbatim expression. Further, this expression will be interpolated if requested. This was the case in Ex.1 where `verb_false` was interpolated.

If the condition's length is greater than 1, then each logical values equal to `TRUE` is replaced by `verb_true`, and `FALSE` or `NA` values are replaced with `verb_false`. Note, importantly, that **no interpolation is perfomed in that case**.

Ex.2: `x = 1:3 ; string_magic("x is {&x == 2 ; two ; not two}")` leads to the vector `c("x is not two", "x is two", "x is not two")`.

In that example, when x is odd, it is replaced with "odd", and when even it is replaced with the elements of y.

Using the two ampersand operator ("&&") is like the simple ampersand version but the default for `verb_false` is the variable used in the condition itself. So the syntax is `{&&cond ; verb_true}` and *it does not accept* `verb_false`.

Ex.3: `i = 3 ; string_magic("i = {&&i == 3 ; three}")` leads to "i = three", and to "i = 5" if `i = 5`.

Pluralization:

There is advanced support for pluralization which greatly facilitates the writing of messages in natural language.

There are two ways to pluralize: over length or over value. To trigger a "pluralization" interpolation use as first character:

- `$` to pluralize over the length of a variable (see Ex.2)
- `#` to pluralize over the value of a variable (see Ex.1)

Ex.1: `x = 5; string_magic("I bought {N?x} book{#s}.")` leads to "I bought five books.". If `x = 1`, this leads to "I bought one book.".

The syntax is `{#plural_ops ? variable}` or `{#plural_ops}` where `plural_ops` are specific pluralization operations which will be described below. The pluralization is perfomed *always* with respect to the value of a variable. You can either add the variable explicitly (`{#plural_ops ? variable}`) or refer to it implicitly (`{#plural_ops}`). If implicit, then the algorithm will look at the previous variable that was interpolated and pluralize over it. This is exaclty what happens in Ex.1 where `x` was interpolated in `{N?x}` and plural operation `s` in `{#s}` then applied to `x`. It was equivalent to have `{#s ? x}`. If a variable wasn't interpolated before, then the next interpolated variable will be used (see Ex.2). If no variable is interpolated at all, an error is thrown.

Ex.2: `x = c("J.", "M."); string_magic("My BFF{$s, are} {C?x}!")` leads to "My BFFs are J. and M.!". If "x = "S.", this leads to "My BFF is S.!".

Pluralizing accepts the following operations:

- s, es: adds an "s" (or "es") if it is plural (> 1), nothing otherwise. Accepts the option `0` or `zero` which treats a 0-length or a 0-value as plural.
- y or ies: adds an 'y' if singular and 'ies' if plural (>1). Accepts the option `0` or `zero` which treats a 0-length or a 0-value as plural.
- enum: enumerates the elements (see help for the regular operation `enum`)
- n, N, len, Len: add the number of elements ("len") or the value ("n") of the variable as a formatted number or in letters (upper case versions). Accepts the options `letter` (to write in letter) and `upper` (to uppercase the first letter).
- nth, ntimes: writes the value of the variable as an order (nth) or a frequence (ntimes). Accepts the option `letter` to write the numbers in letters (uppercase version of the operator does the same).
- is, or any verb: conjugates the verb appropriately

You can chain operations, in that case a whitespace is automatically added between them.

Ex.3: `x = c(7, 3, 18); string_magic("The winning number{$s, is, enum ? sort(x)}.")` leads to "The winning numbers are 3, 7 and 18.". With `x = 7` this leads to "The winning number is 7.".

On top of the previous operations, there is a special operation allowing to add verbatim text depending on the situation. The syntax is as follows:

- `(s1;s2)`: adds verbatim 's1' if singular and 's2' if plural (>1)
- `(s1;s2;s3)`: adds verbatim 's1' if zero, 's2' if singular (=1) and 's3' if plural
- `(s1;;s3)`: adds verbatim 's1' if zero, 's3' if singular or plural (i.e. >=1)

These case-dependent verbatim values **are interpolated** (if appropriate). In these interpolations you need not refer explicitly to the variable for pluralization interpolations.

Ex.4: `x = 3; string_magic("{#(Sorry, nothing found.;;{#N.upper} match{#es, were} found.)?x}")` leads to "Three matches were found.". If "x = 1", this leads to "One match was found." and if "x = 0" this leads to "Sorry, nothing found.".

Escaping and special cases

The opening and closing brakets, `{}`, are special characters and cannot be used as regular text. To bypass their special meaning, you need to escape them with a double backslash.

Ex.1: `string_magic("open = \\\\{, close = }")` leads to `"open = {, close = }"`. Ex.2: `string_magic("many {5 times.c ! \\\\}}")` leads to `many }}}}}`.

You only need to escape the special delimiters which the algorithm is currently looking for. As you can see, you don't need to escape the closing bracket in Ex.1 since no box was open. On the other hand, you need to escape it in Ex.2.

Alternatively, use the argument `.delim` to set custom delimiters.

Ex.3: `string_magic("I {'can {write} {{what}} I want'}")` leads to `"I can {write} {{what}} I want"`.

Since `{expr}` evaluates `expr`, the stuff inside the *box*, you can pass a character string and it will stay untouched.

In the few operations expecting a semi-colon (if-else and pluralization), it can also be escaped with a double backslash.

In interpolations, the exclamation mark (`!`) signals a verbatim expression. But what if you use it to mean the logical operation *not* in an operation-free interpolation? In that case, you need a hack: use a question mark (`?`) first to indicate to the algorithm that you want to evaluate the expression.

Ex.4: `string_magic("{!TRUE} is {?!TRUE}")` leads to "TRUE is FALSE". The first expression is taken verbatim while the second is evaluated.

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (`/`). Example: `string_which(c("hello...", "world"), "fixed/.")` returns `1`. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion `string_which(c("hello...", "world"), ".")` returns `1:2`.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries (`"\\b"` in regex language) to the pattern. Further, the comma (`","`) becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: `string_clean("Am I ambushed?", "wi/am")` leads to " I ambushed?" thanks to the flags "ignore" and "word".
- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if `letters = "aiou"` then `string_clean("My great goose!", "magic/[{letters}] => e")` leads to `"My greet geese!"`

See Also

Other tools with aliases: `cat_magic_alias()`, `string_clean_alias()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
#
# BASIC USAGE ####
#

x = c("Romeo", "Juliet")

# {x} inserts x
string_magic("Hello {x}!")

# elements in ... are collapsed with "" (default)
string_magic("Hello {x[1]}, ",
    "how is {x[2]} doing?")

# Splitting a comma separated string
# The mechanism is explained later
string_vec("J. Mills, David, Agnes, Dr Strong")

# Nota: this is equivalent to (explained later)
string_magic("{', *'S ! J. Mills, David, Agnes, Dr Strong}")

#
# Applying low level operations to strings
#

# Two main syntax:

# A) expression evaluation
# {operation ? x}
#            | |
#            |  \-> the expression to be evaluated
#             \-> ? means that the expression will be evaluated

# B) verbatim
# {operation ! x}
#            | |
#            |  \-> the expression taken as verbatim (here 'x')
#             \-> ! means that the expression is taken as verbatim

# operation: usually 'arg'op with op an operation code.

# Example: splitting
x = "hello dear"
string_magic("{' 's ? x}")
# x is split by ' '

string_magic("{' 's ! hello dear}")
# 'hello dear' is split by ' '
# had we used ?, there would have been an error


# There are 50+ string operators
# Operators usually have a default value
# Operations can have options
# Operations can be chained by separating them with a comma

# Example: default of 's' is ' ' + chaining with collapse
string_magic("{s, ' my 'c ! hello dear}")

#
# Nesting
#

# {operations ! s1{expr}s2}
#             |   |
#             |    \-> expr will be interpolated then added to the string
#              \-> nesting requires verbatim evaluation: '!'

string_magic("The variables are: {C ! x{1:4}}.")

# This one is ugly but it shows triple nesting
string_magic("The variables are: {ws, C ! {2 times ! x{1:4}}{','s, 4 each !  ,_sq}}.")

#
# Splitting
#

# s: split with fixed pattern, default is ' '
string_magic("{s ! a b c}")
string_magic("{' b 's !a b c}")

# S: same as 's' but default is ',[ \t\n]*'
string_magic("{S !a, b, c}")
string_magic("{'[[:punct:] ]+'S ! a! b; c}")

# add regex flags: e.g. fixed search
string_magic("{'f/.'s ! hi.there}")


#
# Collapsing
#

# c and C do the same, their default is different
# syntax: 's1|s2' with
# - s1 the string used for collapsing
# - s2 (optional) the string used for the last collapse

# c: default is ' '
string_magic("{c ? 1:3}")

# C: default is ', | and '
string_magic("{C ? 1:3}")

string_magic("{', | or 'c ? 1:4}")
```

```
#
# Extraction
#

# extract: to extract patterns (option first)
# x: alias to extract.first
# X: alias to extract
# syntax: 'pattern'x
# Default is '[[:alnum:]]+'

x = "This years is... 2020"
string_magic("{x ? x}") # similar to string_magic("{extract.first ? x}")
string_magic("{X ? x}") # similar to string_magic("{extract ? x}")

string_magic("{'\\d+'x ? x}")

#
# STRING FORMATTING ####
#

#
# upper, lower, title

# upper case the first letter
string_magic("{upper.first ! julia mills}")

# title case
string_magic("{title ! julia mills}")

# upper all letters
string_magic("{upper ! julia mills}")

# lower case
string_magic("{lower ! JULIA MILLS}")

#
# q, Q, bq: single, double, back quote

string_magic("{S, q, C ! Julia, David, Wilkins}")
string_magic("{S, Q, C ! Julia, David, Wilkins}")
string_magic("{S, bq, C ! Julia, David, Wilkins}")

#
# format, Format: formats the string to fit the same length

# format: the right side is filled with blanks
# Format: the left side is filled with blanks, the string is right aligned

score = c(-10, 2050)
nm = c("Wilkins", "David")
string_magic("Monopoly scores:\n{'\n'c ! - {format ? nm}: {Format ? score} US$}")
```

```
# OK that example may have been a bit too complex,
# let's make it simple:

string_magic("Scores: {format ? score}")
string_magic("Names: {Format ? nm}")

#
# ws: white space normalization

# ws: suppresses trimming white spaces + normalizes successive white spaces
# Add the following options in any order to:
# - punct: remove punctuation
# - digit: remove digits
# - isolated: remove isolated characters

string_magic("{ws ! The   white  spaces are now clean.  }")

string_magic("{ws.punct ! I, really -- truly; love punctuation!!!}")

string_magic("{ws.digit ! 1, 2, 12, a microphone check!}")

string_magic("{ws.i ! 1, 2, 12, a microphone check!}")

string_magic("{ws.d.i ! 1, 2, 12, a microphone check!}")

string_magic("{ws.p.d.i ! 1, 2, 12, a microphone check!}")

#
# %: applies sprintf formatting

 # add the formatting as a regular argument
string_magic("pi = {'.2f'% ? pi}")
# or right after the %
string_magic("pi = {%.2f ? pi}")

#
# paste: appends text on each element
# Accepts the options: right, both, front and back
# It accepts the special values :1:, :i:, :I:, :a:, :A: to create enumerations

# adding '|' on both sides
string_magic("{'|'paste.both, ' + 'c ! x{1:4}}")


# Enumerations
acad = string_vec("you like admin, you enjoy working on weekends, you really love emails")
string_magic("Main reasons to pursue an academic career:\n {':i:) 'paste, C ? acad}.")

# You can also use the enum command
string_magic("Main reasons to pursue an academic career:\n {enum.i ? acad}.")

#
# stopwords: removes basic English stopwords
# the list is from the Snowball project:
#  http://snowball.tartarus.org/algorithms/english/stop.txt

string_magic("{stop, ws ! It is a tale told by an idiot, ",
                         "full of sound and fury, signifying nothing.}")

#
# k: keeps the first n characters
# syntax: nk: keeps the first n characters
#         'n|s'k: same + adds 's' at the end of shortened strings
#         'n||s'k: same but 's' counts in the n characters kept

words = string_vec("short, constitutional")
string_magic("{5k ? words}")

string_magic("{'5|..'k ? words}")

string_magic("{'5||..'k ? words}")

#
# K: keeps the first n elements
# syntax: nK: keeps the first n elements
#         'n|s'K: same + adds the element 's' at the end
#         'n||s'K: same but 's' counts in the n elements kept
#
# Special values :rest: and :REST:, give the number of items dropped

bx = string_vec("Pessac Leognan, Saint Emilion, Marguaux, Saint Julien, Pauillac")
string_magic("Bordeaux wines I like: {3K, ', 'C ? bx}.")

string_magic("Bordeaux wines I like: {'3|etc..'K, ', 'C ? bx}.")

string_magic("Bordeaux wines I like: {'3||etc..'K, ', 'C ? bx}.")

string_magic("Bordeaux wines I like: {'3|and at least :REST: others'K, ', 'C ? bx}.")

#
# Ko, KO: special operator which keeps the first n elements and adds "others"
# syntax: nKo
# KO gives the rest in letters

string_magic("Bordeaux wines I like: {4KO, C ? bx}.")

#
# r, R: string replacement
# syntax: 's'R: deletes the content in 's' (replaces with the empty string)
#         's1 => s2'R replaces s1 into s2

string_magic("{'e'r, ws ! The letter e is deleted}")

# adding a perl look-behind
string_magic("{'(?<! )e'r !The letter e is deleted}")
string_magic("{'e => a'r !The letter e becomes a}")

string_magic("{'([[:alpha:]]{3})[[:alpha:]]+ => \\1.'r ! Trimming the words}")

# Alternative way with simple operations: split, shorten, collapse
string_magic("{s, '3|.'k, c ! Trimming the words}")

#
# times, each
# They accept the option c to collapse with the empty string

string_magic("N{10 times.c ! o}!")

string_magic("{3 times.c ? 1:3}")
string_magic("{3 each.c ? 1:3}")

#
# erase: replaces the items by the empty string
# -> useful in conditions

string_magic("{erase ! I am going to be annihilated}")

#
# ELEMENT MANIPULATION ####
#

#
# rm: removes the elements
# Its (optional) argument is a regular expression giving which element to remove
# Many options: "empty", "blank", "noalpha", "noalnum", "all"

x = c("Destroy", "All")
string_magic("{'A'rm ? x}")

string_magic("{rm.all ? x}")

x = string_vec("1, 12, 123, 1234, 123456, 1234567")
# we delete elements whose number of characters is lower or equal to 3
# => see later section CONDITIONS
string_magic("{if(.nchar > 3 ; nuke) ? x}")

#
# PLURALIZATION ####
#

# Two ways to enable pluralization:
# {$ command}: means the plural is decuced from the length of the variable
# {# command}: means the plural is decuced from the value of the variable

# Explanatory example
x = c("Eschyle", "Sophocle", "Euripide")
n = 37
string_magic("The author{$s, enum, have ? x} written {#N ? n} play{#s}.")
```

```
x = "Laurent Berge"
n = 0
string_magic("The author{$s, enum, have ? x} written {#N ? n} play{#s}.")

# How does it work?
# First is {$s, enum, have ? x}.
# The commands `s`, `enum` and `have` are applied to `x` which must come after a `?`
#    => there the plural (whether an s is added and how to conjugate the verb have) depends
#       on the **length** of the vector `x`
#
# Second comes {#N ? n}.
# The double dollar sign means that the command `N` will be applied to the **value** n.
# The value must come after the `?`
#
# Third is {#s}.
# The object to which `s` should be applied is missing (there is no `? n`).
# The default is to apply the command to the previous object. In this case,
#  this is `n`.

# Another similar example illustrating that we need not express the object several times:
x = c("Eschyle", "Sophocle", "Euripide")
string_magic("The {Len ? x} classic author{$s, are, enum}.")



#
# ARGUMENTS FROM THE ENVIRONMENT ####
#

# Arguments can be evaluated from the calling environment.
# Simply use backticks instead of quotes.

dollar = 6
reason = "glory"
string_magic("Why do you develop packages? For {`dollar`times.c ! $}?",
    "For money? No... for {upper,''s, c ? reason}!", .sep = "\n")

#
# Alias generation
#

# Let's create a formula filler
# - we use .local_ops to create the ad hoc operation "add" which adds variables
# - we transform into a formula ex post

fml = string_magic_alias(.post = as.formula, .local_ops = list(add = "' + 'collapse"))

# example with mtcars
lhs = "mpg"
rhs = c("hp", "drat")
fml("{lhs} ~ {add?rhs} + am")
```

`string_magic_alias` *Create* `string_magic` *aliases with custom defaults*

Description

Utility to easily create `string_magic` aliases with custom default

Usage

```
string_magic_alias(
  .sep = "",
  .vectorize = FALSE,
  .delim = c("{", "}"),
  .last = NULL,
  .post = NULL,
  .default = NULL,
  .nest = FALSE,
  .invisible = FALSE,
  .local_ops = NULL,
  .collapse = NULL,
  .check = TRUE,
  .class = NULL,
  .namespace = NULL
)
```

Arguments

`.sep`	Character scalar, default is the empty string `""`. It is used to collapse all the elements in `...` before applying any operation.
`.vectorize`	Logical scalar, default is `FALSE`. If `TRUE`, Further, elements in `...` are NOT collapsed together, but instead vectorised.
`.delim`	Character vector of length 1 or 2. Default is `c("{", "}")`. Defines the opening and the closing delimiters for interpolation. If of length 1, it must be of the form: 1) the opening delimiter, 2) a single space, 3) the closing delimiter. Ex: `".[ ]"` is equivalent to `c(".[", "]")`. The default value is equivalent to `"{ }"`. [ ]: R:%20 [", "]: R:%22,%20%22
`.last`	Character scalar, a function, or `NULL` (default). If provided and character: it must be an `string_magic` chain of operations of the form `"'arg1'op1, op2, etc"`. All these operations are applied just before returning the vector. If a function, it will be applied to the resulting vector.

`.post`	Function or `NULL` (default). If not `NULL`, this function will be applied after all the processing, just before returning the object. This function can have extra arguments which will be caught directly in the `...` argument of `string_magic`. For example if `.post = head`, you can directly pass the argument `n = 3` to `string_magic`'s arguments.
`.default`	Character scalar or `NULL` (default). If provided, it must be a sequence of `string_magic` operations. It will be applied as a default to any interpolation. Ex: if `x = 1:2`, then `string_magic("x = {x}", .default = "enum")` leads to "x = 1 and 2", and is equivalent to `string_magic("x = {enum?x}")`. Note that this default operations does not apply to nested expressions. That is `string_magic("{!x{1:2}}", .default = "enum")` leads to `c("x1", "x2")` and NOT `"x1 and 2"`.
`.nest`	Logical, default is `FALSE`. If `TRUE`, it will nest the original string within interpolation delimiters, so that you can apply operations directly on the string. Example: `string_magic("upper ! hello")` returns "upper ! hello", while `string_magic("upper ! hello", .nest = TRUE)` returns `"HELLO"`.
`.invisible`	Logical scalar, default is `FALSE`. Whether the object returned should be invisible (i.e. not printed on the console).
`.local_ops`	Named list or `NULL` (default). If provided, it must be a list of the form `list(alias1 = ops1, alias2 = ops2)` where alias is the name of the newly defined operator an ops is a character scalar representing the associated string_magic operations. Ex: `list(add = "' + 'collapse")` creates the operation add which collapses the string with pluses. All operations created here are only available to the generated function.
`.collapse`	Character scalar, default is `NULL`. If provided, the character vector that should be returned is collapsed with the value of this argument. This leads to return a string of length 1.
`.check`	Logical scalar, default is `TRUE`. Whether to enable error-handling (i.e. human readable error messages). Without error-handling you can save something of the order of 40us. Useful only in long loops.
`.class`	Character vector representing the class to give to the object returned. By default it is `NULL`. Note that the class `string_magic` has a specific `print` method, usually nicer for small vectors (it `base::cat()`s the elements).
`.namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`.

Details

Use this function if you want to change `string_magic` default values. For example, if you want the interpolation to be done with `"{{}}"` (instead of `{}`) or if you want the default separation to be the space (instead of the empty string). See the example.

Value

This function returns a function which will behave in the same way as `string_magic()`

Writing a package using `string_magic`

If you want to use `string_magic` in your package and want to make use of custom operations:

- place any `string_magic_register_fun` and `string_magic_register_ops` in your `.onLoad` function (see `help("onLoad")`). The .onLoad function is run whenever the package is loaded for the first time. It's a function that you can place anywhere in your `R/*` files and which looks like this:

```
.onLoad = function(libname, pkgname){
  # string_magic custom operations
  string_magic_register_ops("'80|-'fill", "h1")

  invisible()
}
```

- if you don't want to place the `string_magic_register_*` functions in the .onLoad function, you can, but then you **must** provide the argument `namespace`:

```
string_magic_register_ops("'80|-'fill", "h1", namespace = "myPackageName")
```

- you must create an `string_magic_alias()` to create an alias to `string_magic()` and use the argument `.namespace = "myPackageName"`. Use this opportunity to change the defaults if you wish. You can even override the `string_magic` function:

```
# creating an alias with the same name + changing the delimiter
string_magic = stringmagic::string_magic_alias(.namespace = "myPackageName", .delim = "{{ }}")
```

Author(s)

Laurent Berge

See Also

Other related to string_magic: `string_magic_register_fun()`

Other tools with aliases: `cat_magic_alias()`, `string_clean_alias()`, `string_magic()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
# we create the function sma2 with different defaults
sma2 = string_magic_alias(.delim = ".[ ]", .sep = " ", .class = "string_magic")

person = "john doe"
sma2("Hello", ".[title ? person]")

# you can use the arguments whose default has been changed
sma2("Hello", ".[title ? person]", .sep = ": ")
```

string_magic_register_fun *Register custom operations to apply them in* `string_magic`

Description

Extends the capabilities of `string_magic()` by adding any custom operation

Usage

```
string_magic_register_fun(fun, alias, valid_options = NULL, namespace = NULL)

string_magic_register_ops(ops, alias, namespace = NULL)
```

Arguments

`fun`	A function which must have at least the arguments 'x' and '...'. Additionnaly, it can have the arguments: 'argument', 'options', 'group', 'group_flag'. This function must return a vector. This function will be internally called by `string_magic` in the form `fun(x, argument, options, group, group_flag)`.`x`: the value to which the operation applies. `argument`: the quoted `string_magic` argument (always character). `options`: a character vector of `string_magic` options. The two last arguments are of use only in group-wise operations if `fun` changes the lengths of vectors. `group`: an index of the group to which belongs each observation (integer). `group_flag`: value between 0 and 2; 0: no grouping operation requested; 1: keep track of groups; 2: apply grouping.
`alias`	Character scalar, the name of the operation.
`valid_options`	A character vector or NULL (default). Represents a list of valid options for the operation. This is used: a) to enable auto-completion, b) for error-handling purposes.
`namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. If the function `string_magic_register_*` is located in the `onLoad` function (see `help("onLoad")`), there is nothing to do. Otherwise, pass the name of your package in this argument to make all the new operation definitions scoped (i.e. only your package can access it and it can't be messed up by end users).
`ops`	Character scalar representing a valid chain of `string_magic` operations. It should be of the form `"op1, 'arg'op2, etc"`. For example `"'80|-'fill"` fills the line with dashes up to 80 characters.

Details

We try to strongly check the new operations since it's always better to find out problems sooner than later. This means that when the function is defined, it is also tested.

If you pass a function, note that it should work for non-character arguments in `x`.

Value

These function do not return anything. They register new operations to be used in the `string_magic` family of functions by placing them in the options (later fetched by `string_magic()` at run-time).

Functions

- `string_magic_register_ops()`: Create new combinations of `string_magic` operations

Writing a package using `string_magic`

If you want to use `string_magic` in your package and want to make use of custom operations:

- place any `string_magic_register_fun` and `string_magic_register_ops` in your `.onLoad` function (see `help("onLoad")`). The .onLoad function is run whenever the package is loaded for the first time. It's a function that you can place anywhere in your `R/*` files and which looks like this:

```
.onLoad = function(libname, pkgname){
  # string_magic custom operations
  string_magic_register_ops("'80|-'fill", "h1")

  invisible()
}
```

- if you don't want to place the `string_magic_register_*` functions in the .onLoad function, you can, but then you **must** provide the argument `namespace`:

```
string_magic_register_ops("'80|-'fill", "h1", namespace = "myPackageName")
```

- you must create an `string_magic_alias()` to create an alias to `string_magic()` and use the argument `.namespace = "myPackageName"`. Use this opportunity to change the defaults if you wish. You can even override the `string_magic` function:

```
# creating an alias with the same name + changing the delimiter
string_magic = stringmagic::string_magic_alias(.namespace = "myPackageName", .delim = "{{ }}")
```

Author(s)

Laurent R. Berge

See Also

Other related to string_magic: `string_magic_alias()`

Examples

```
# let's create an operation that adds markdown emphasis
# to elements of a vector

# A) define the function
fun_emph = function(x, ...) paste0("*", x, "*")

# B) register it
string_magic_register_fun(fun_emph, "emph")

# C) use it
x = string_vec("right, now")
string_magic("Take heed, {emph, c? x}.")

#
# now let's add the option "strong"
fun_emph = function(x, options, ...) {
  if("strong" %in% options){
    paste0("***", x, "***")
  } else {
    paste0("*", x, "*")
  }
}

string_magic_register_fun(fun_emph, "emph", "strong")

x = string_vec("right, now")
string_magic("Take heed, {emph.strong, c? x}.")

#
# now let's add an argument
fun_emph = function(x, argument, options, ...){
  arg = argument
  if(nchar(arg) == 0) arg = "*"

  if("strong" %in% options){
    arg = paste0(rep(arg, 3), collapse = "")
  }

  paste0(arg, x, arg)
}

string_magic_register_fun(fun_emph, "emph", "strong")

x = string_vec("right, now")
string_magic("Take heed, {'_'emph.s, c? x}.")

#
# using string_magic_register_ops
#

# create a 'header' maker
```

```
string_magic_register_ops("tws, '# 'paste, ' 'paste.right, '40|-'fill", "h1")
cat_magic("{h1 ! My title}\n my content")
```

string_ops_alias — *Chains basic operations to character vectors*

Description

Simple tool to perform multiple operations to character vectors.

Usage

```
string_ops_alias(op = NULL, pre_unik = NULL, namespace = NULL)

string_ops(
  x,
  ...,
  op = NULL,
  pre_unik = NULL,
  namespace = NULL,
  envir = parent.frame()
)

stops(
  x,
  ...,
  op = NULL,
  pre_unik = NULL,
  namespace = NULL,
  envir = parent.frame()
)
```

Arguments

`op`	Character **vector** or `NULL` (default). Character scalar containing the comma separated values of operations to perform to the vector. The 50+ operations are detailed in the help page of `string_magic()`. Note that if this argument is provided, then the values in ... are ignored.
`pre_unik`	Logical scalar, default is `NULL`. Whether to first unique the vector before applying the possibly costly string operations, and merging back the result. For very large vectors with repeated values the time gained can be substantial. By default, this is `TRUE` for vector of length 1M or more.

`namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`. If so pass the name of your package in this argument so that your function can access the new `string_magic` operations defined within your package.
`x`	A character vector. If not a character vector but atomistic (i.e. not a list), it will be converted to a character vector.
`...`	Character **scalars**. Character scalar containing the comma separated values of operations to perform to the vector. The 50+ operations are detailed in the help page of `string_magic()`.
`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.

Details

This function is a simple wrapper around string_magic. Formally, `string_ops(x, "op1, op2")` is equivalent to `string_magic("{op1, op2 ? x}")`.

Value

In general it returns a character vector. It may be of a length different from the original one, depending on the operations performed.

Functions

- `string_ops_alias()`: `string_ops` alias with custom defaults
- `stops()`: Alias to `string_ops`

Author(s)

Laurent R. Berge

See Also

Other tools with aliases: `cat_magic_alias()`, `string_clean_alias()`, `string_magic()`, `string_magic_alias()`, `string_vec_alias()`

Examples

```
# data on car models
cars = row.names(mtcars)

# let's get the brands starting with an "m"
string_ops(cars, "'i/^m'get, x, unik")

# Explainer:
# 'i/^m'get: keeps only the elements starting with an m,
#            i/ is the 'regex-flag' "ignore" to ignore the case
#            ^m means "starts with an m" in regex language
# x: extracts the first pattern. The default pattern is "[[:alnum:]]+"
#    which means an alpha-numeric word
# unik: applies unique() to the vector
# => see help in ?string_magic for more details on the operations


# let's get the 3 largest numbers appearing in the car models
string_ops(cars, "'\\d+'x, rm, unik, num, dsort, 3 first")

# Explainer:
# '\d+'x: extracts the first pattern, the pattern meaning "a succession"
#          of digits in regex language
# rm: removes elements equal to the empty string (default behavior)
# unik: applies unique() to the vector
# num: converts to numeric
# dsort: sorts in decreasing order
# 3 first: keeps only the first three elements

# You can use several character vectors as operations:
string_ops(cars,
           "'\\d+'x, rm, unik",
           "num, dsort, 3 first")
```

string_split — *Splits a character string wrt a pattern*

Description

Splits a character string with respect to pattern

Usage

```
string_split(
  x,
  split,
  simplify = TRUE,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  envir = parent.frame()
)

stsplit(
  x,
  split,
  simplify = TRUE,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  envir = parent.frame()
)
```

Arguments

`x`	A character vector.
`split`	A character scalar. Used to split the character vectors. By default this is a regular expression. You can use flags in the pattern in the form `flag1, flag2/pattern`. Available flags are `ignore` (case), `fixed` (no regex), word (add word boundaries), magic (add interpolation with `"{}"`). Example: if "ignore/hello" and the text contains "Hello", it will be split at "Hello". Shortcut: use the first letters of the flags. Ex: "iw/one" will split at the word "one" (flags 'ignore' + 'word').
`simplify`	Logical scalar, default is `TRUE`. If `TRUE`, then when the vector input `x` is of length 1, a character vector is returned instead of a list.
`fixed`	Logical, default is `FALSE`. Whether to consider the argument `split` as fixed (and not as a regular expression).
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then pattern = "The, mountain" will select strings containing either the word 'The' or the word 'mountain'.
`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.

Value

If `simplify = TRUE` (default), the object returned is:

- a character vector if `x`, the vector in input, is of length 1: the character vector contains the result of the split.
- a list of the same length as `x`. The ith element of the list is a character vector containing the result of the split of the ith element of `x`.

If `simplify = FALSE`, the object returned is always a list.

Functions

- `stsplit()`: Alias to `string_split`

Generic regular expression flags

All `stringmagic` functions support generic flags in regular-expression patterns. The flags are useful to quickly give extra instructions, similarly to *usual* regular expression flags.

Here the syntax is "flag1, flag2/pattern". That is: flags are a comma separated list of flag-names separated from the pattern with a slash (`/`). Example: `string_which(c("hello...", "world"),` "fixed/.") returns 1. Here the flag "fixed" removes the regular expression meaning of "." which would have otherwise meant *"any character"*. The no-flag verion string_which(c("hello...", "world"), ".") returns 1:2.

Alternatively, and this is recommended, you can collate the initials of the flags instead of using a comma separated list. For example: "if/dt[" will apply the flags "ignore" and "fixed" to the pattern "dt[".

The four flags always available are: "ignore", "fixed", "word" and "magic".

- "ignore" instructs to ignore the case. Technically, it adds the perl-flag "(?i)" at the beginning of the pattern.
- "fixed" removes the regular expression interpretation, so that the characters ".", "$", "^", "[" (among others) lose their special meaning and are treated for what they are: simple characters.
- "word" adds word boundaries ("\\b" in regex language) to the pattern. Further, the comma (",") becomes a word separator. Technically, "word/one, two" is treated as "\b(one|two)\b". Example: string_clean("Am I ambushed?", "wi/am") leads to " I ambushed?" thanks to the flags "ignore" and "word".
- "magic" allows to interpolate variables inside the pattern before regex interpretation. For example if letters = "aiou" then string_clean("My great goose!", "magic/[{letters}] => e") leads to "My greet geese!"

Examples

```
time = "This is the year 2024."

# we break the sentence
string_split(time, " ")

# simplify = FALSE leads to a list
string_split(time, " ", simplify = FALSE)

# let's break at "is"
string_split(time, "is")

# now breaking at the word "is"
# NOTE: we use the flag `word` (`w/`)
string_split(time, "w/is")

# same but using a pattern from a variable
# NOTE: we use the `magic` flag
pat = "is"
string_split(time, "mw/{pat}")
```

`string_split2df` *Splits a character vector into a data frame*

Description

Splits a character vector and formats the resulting substrings into a data.frame

Usage

```
string_split2df(
  x,
  data = NULL,
  split = NULL,
  id = NULL,
  add.pos = FALSE,
  id_unik = TRUE,
  fixed = FALSE,
  ignore.case = FALSE,
  word = FALSE,
  envir = parent.frame(),
  dt = FALSE,
  ...
)

string_split2dt(
  x,
  data = NULL,
  split = NULL,
  id = NULL,
  add.pos = FALSE,
  id_unik = TRUE,
  fixed = FALSE
)
```

Arguments

`x`	A character vector or a two-sided formula. If a two-sided formula, then the argument `data` must be provided since the variables will be fetched in there. A formula is of the form `char_var ~ id1 + id2` where `char_var` on the left is a character variable and on the right `id1` and `id2` are identifiers which will be included in the resulting table. Alternatively, you can provide identifiers via the argument `id`.
`data`	Optional, only used if the argument `x` is a formula. It should contain the variables of the formula.
`split`	A character scalar. Used to split the character vectors. By default this is a regular expression. You can use flags in the pattern in the form `flag1, flag2/pattern`.

	Available flags are `ignore` (case), `fixed` (no regex), word (add word boundaries), magic (add interpolation with `"{}"`). Example: if "ignore/hello" and the text contains "Hello", it will be split at "Hello". Shortcut: use the first letters of the flags. Ex: "iw/one" will split at the word "one" (flags 'ignore' + 'word').
`id`	Optional. A character vector or a list of vectors. If provided, the values of `id` are considered as identifiers that will be included in the resulting table.
`add.pos`	Logical, default is `FALSE`. Whether to include the position of each split element.
`id_unik`	Logical, default is `TRUE`. In the case identifiers are provided, whether to trigger a message if the identifiers are not unique. Indeed, if the identifiers are not unique, it is not possible to reconstruct the original texts.
`fixed`	Logical, default is `FALSE`. Whether to consider the argument `split` as fixed (and not as a regular expression).
`ignore.case`	Logical scalar, default is `FALSE`. If `TRUE`, then case insensitive search is triggered.
`word`	Logical scalar, default is `FALSE`. If `TRUE` then a) word boundaries are added to the pattern, and b) patterns can be chained by separating them with a comma, they are combined with an OR logical operation. Example: if `word = TRUE`, then pattern = "The, mountain" will select strings containing either the word 'The' or the word 'mountain'.
`envir`	Environment in which to evaluate the interpolations if the flag `"magic"` is provided. Default is `parent.frame()`.
`dt`	Logical, default is `FALSE`. Whether to return a `data.table`. See also the function `string_split2dt`.
`...`	Not currently used.

Value

It returns a `data.frame` or a `data.table` which will contain: i) obs: the observation index, ii) pos: the position of the text element in the initial string (optional, via add.pos), iii) the text element, iv) the identifier(s) (optional, only if `id` was provided).

Functions

- `string_split2dt()`: Splits a string vector and returns a `data.table`

See Also

String operations: `string_is()`, `string_get()`, `string_clean()`, `string_split2df()`. Chain basic operations with `string_ops()`. Clean character vectors efficiently with `string_clean()`.

Use `string_vec()` to create simple string vectors.

String interpolation combined with operation chaining: `string_magic()`. You can change `string_magic` default values with `string_magic_alias()` and add custom operations with `string_magic_register_fun()`.

Display messages while benefiting from `string_magic` interpolation with `cat_magic()` and `message_magic()`.

Other tools with aliases: `cat_magic_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
x = c("Nor rain, wind, thunder, fire are my daughters.",
      "When my information changes, I alter my conclusions.")

id = c("ws", "jmk")

# we split at each word
string_split2df(x, "[[:punct:] ]+")

# we add the 'id'
string_split2df(x, "[[:punct:] ]+", id = id)

# TO NOTE:
# - the second argument is `data`
# - when it is missing, the argument `split` becomes implicitly the second
# - ex: above we did not use `split = "[[:punct:] ]+"`

#
# using the formula

base = data.frame(text = x, my_id = id)
string_split2df(text ~ my_id, base, "[[:punct:] ]+")

#
# with 2+ identifiers

base = within(mtcars, carname <- rownames(mtcars))

# we have a message because the identifiers are not unique
string_split2df(carname ~ am + gear + carb, base, " +")

# adding the position of the words & removing the message
string_split2df(carname ~ am + gear + carb, base, " +", id_unik = FALSE, add.pos = TRUE)
```

`string_vec_alias` *Efficient creation of string vectors with optional interpolation*

Description

Create string vectors in multiple ways: 1) add successive string elements (like in `c()`), or 2) write a character string that will be broken with respect to commas (`"hi, there"` becomes `c("hi", "there")`), or 3) interpolate variables in character strings (`"x{1:2}"` becomes `c("x1", "x2")`) with full access to `string_magic()` operations, or any combination of the three.

Usage

```
string_vec_alias(
    .cmat = FALSE,
    .nmat = FALSE,
    .df = FALSE,
    .df.convert = TRUE,
    .last = NULL,
    .delim = c("{", "}"),
    .split = TRUE,
    .protect.vars = FALSE,
    .check = TRUE,
    .sep = NULL,
    .collapse = NULL,
    .namespace = NULL
  )

  string_vec(
    ...,
    .cmat = FALSE,
    .nmat = FALSE,
    .df = FALSE,
    .df.convert = TRUE,
    .delim = c("{", "}"),
    .envir = parent.frame(),
    .split = TRUE,
    .protect.vars = FALSE,
    .sep = NULL,
    .last = NULL,
    .check = TRUE,
    .help = NULL,
    .collapse = NULL,
    .namespace = NULL
  )

  stvec(
    ...,
    .cmat = FALSE,
    .nmat = FALSE,
    .df = FALSE,
    .df.convert = TRUE,
    .delim = c("{", "}"),
    .envir = parent.frame(),
    .split = TRUE,
    .protect.vars = FALSE,
    .sep = NULL,
    .last = NULL,
    .check = TRUE,
    .help = NULL,
    .collapse = NULL,
    .namespace = NULL
```

```
)
```

Arguments

`.cmat`	Logical scalar (default is `FALSE`), integer or complex with integer values. If `TRUE`, we try to coerce the result into a **character** matrix. The number of rows and columns is deduced from the look of the arguments. An integer indicates the number of rows. If a complex, the imaginary part represents the number of columns. Ex: `5i` means 5 columns, `3 + 2i` means 3 rows and 2 columns. The matrix is always filled by row.
`.nmat`	Logical scalar (default is `FALSE`), integer or complex with integer values. If `TRUE`, we try to coerce the result into a **numeric** matrix. The number of rows and columns is deduced from the look of the arguments. An integer indicates the number of rows. If a complex, the imaginary part represents the number of columns. Ex: `5i` means 5 columns, `3 + 2i` means 3 rows and 2 columns. The matrix is always filled by row. Non numbers are silently turned to NA.
`.df`	Logical scalar (default is `FALSE`), integer, complex with integer values, or character vector. If `TRUE`, we try to coerce the result into a `data.frame`. The number of rows and columns is deduced from the look of the arguments. An integer indicates the number of rows. If a complex, the imaginary part represents the number of columns. Ex: `5i` means 5 columns, `3 + 2i` means 3 rows and 2 columns. If a character vector: it should give the column names. Note that if the vector is of length 1, its values are comma separated (i.e. the value `"id, name"` is identical to `c("id", "name")`). Note that the columns that can be converted to numeric are converted to numeric. The other columns are in string form. Monitor this behavior with `.df.convert`.
`.df.convert`	Logical scalar, default is `TRUE`. Only used when the result is to be converted to a data frame (with the argument `.df`). If `TRUE`, any column looking like a numeric vector is converted to numeric. Otherwise all columns are character strings.
`.last`	Character scalar, a function, or `NULL` (default). If provided and character: it must be an `string_magic` chain of operations of the form `"'arg1'op1, op2, etc"`. All these operations are applied just before returning the vector. If a function, it will be applied to the resulting vector.
`.delim`	Character vector of length 1 or 2. Default is `c("{", "}")`. Defines the opening and the closing delimiters for interpolation. If of length 1, it must be of the form: 1) the opening delimiter, 2) a single space, 3) the closing delimiter. Ex: `".[ ]"` is equivalent to `c(".[", "]")`. The default value is equivalent to `"{ }"`. [ ]: R:%20 [", "]: R:%22,%20%22
`.split`	Logical or a character symbol, default is `TRUE`. If `TRUE`, the character vectors are split with respect to commas (default). If `FALSE`, no splitting is performed. If a character symbol, the string vector will be split according to this symbol. Note that any space after the symbol (including tabs and newlines) is discarded. You can escape the splitting symbol with a backslash right before it. Ex: by default `string_vec("hi, there")` leads to the vector `c("hi", "there")`.

`.protect.vars`	Logical scalar, default is `FALSE`. If `TRUE`, then only arguments equal to a "natural" character scalar are comma-split and interpolated, other arguments are not touched. Ex: `string_vec("x{1:5}")` will lead to a vector of length 5 ("x1" to "x5"), while `z = "x{1:5}"` followed by `string_vec(z)` leads to a vector of length 1: `"x{1:5}"`. If `FALSE`, comma-splitting and interpolation is performed on all variables.
`.check`	Logical scalar, default is `TRUE`. Whether to enable error-handling (i.e. human readable error messages). Without error-handling you can save something of the order of 40us. Useful only in long loops.
`.sep`	Character scalar or `NULL` (default). If not `NULL`, the function `base::paste()` is applied to the resulting vector with `sep = .sep`.
`.collapse`	Character scalar or `NULL` (default). If not `NULL`, the function `base::paste()` is applied to the resulting vector with `collapse = .collapse`. If so, pass the name of your package in this argument so that your function can access the new `string_magic` operations defined within your package.
`.namespace`	Character scalar or `NULL` (default). **Only useful for package developers.** As a regular end-user you shouldn't care! If your package uses `string_magic`, you should care. It is useful **only** if your package uses 'custom' `string_magic` operations, set with `string_magic_register_fun()` or `string_magic_register_ops()`.
`...`	Character vectors that will be vectorized. If commas are present in the character vector, it will be split with respect to commas and following blanks. The vectors can contain any interpolation in the form `"{var}"` and any `string_magic()` operation can be applied. To change the delimiters for interpolation, see `.delim`. Named arguments are used in priority for variable substitution, otherwise the value of the variables to be interpolated are fetched in the calling environment (see argument `.envir`).
`.envir`	An environment used to evaluate the variables in `"{}"`. By default the variables are evaluated using the environment from where the function is called or using the named arguments passed to the function.
`.help`	Character scalar or `TRUE`, default is `NULL`. This argument is used to generate a dynamic help on the console. If `TRUE`, the user can select which topic to read from the main documentation, with the possibility to search for keywords and navigate the help pages. If a character scalar, then a regex search is perfomed on the main documentation and any section containining a match is displayed. The user can easily navigate across matches.

Details

The main objective of this function is to simplify the creation of small character vectors. By default, you can pass a character string of length 1 with values separated with commas and a character vector will be returned.

You can use interpolation using curly brackets (see `string_magic()`). You can pass values for the interpolation directly in the arguments (this is why all arguments start with a dot).

By default character values containing commas are split with respect to the commas to create vectors. To change this behavior, see the argument `.split`.

The default of the argument `.protect.vars` is `FALSE`. To avoid unwanted comma-splitting and interpolations on variables, set it to `TRUE`. The main use case of this function is the creation of small string vectors, which can be written directly at function call.

Customize the default of this function with `string_vec_alias()`.

Value

By default this function returns a string vector, the length of which depends on the arguments.

This result can be processed with the arguments `cmat`, `nmat` and `.df` which will try to coerce the result into a character matrix, a numeric matrix , or a data frame, respectively.

Functions

- `string_vec_alias()`: Create `string_vec` aliases with custom defaults
- `stvec()`: Alias to `string_vec`

Author(s)

Laurent Berge

See Also

Other tools with aliases: `cat_magic_alias()`, `string_clean_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`

Examples

```
# illustrating comma-splitting and interpolation
string_vec("x1, y2, z{4:5}")

# variable protection
x = "x{1:5}"
# without protection (default) => interpolation takes place
string_vec(x, "y{1:2}")

# with protection => no interpolation for x
string_vec(x, "y{1:2}", .protect.vars = TRUE)

# removing comma splitting
string_vec("Hi, said Charles.", "Hi, said {girl}.", girl = "Julia", .split = FALSE)

# changing the delimiters for interpolation
pkg = "\\usepackage[usenames,dvipsnames]{xcolor}"
string_vec("\\usepackage{.[S!graphicx, fourier, standalone]}",
           pkg, .delim = ".[ ]")

#
# Customization
#

# Unhappy about the defaults? Create an alias!
# we create a "matrix generator"
matgen = string_vec_alias(.nmat = TRUE, .last = "'\n'split")

matgen("5, 4, 3
        8, 6, 2")
```

`timer_magic` *Sets up a timer that can be used within `_magic` functions*

Description

Sets up a timer which can later be summoned by `string_magic()` functions via the `.timer`, `.timer_lap` and `.timer_total` variables. Useful to report timings within functions with the function `cat_magic()` or `message_magic()`.

Usage

```
timer_magic()
```

Details

This functions sets up a timer with `base::Sys.time()`. This timer can then be tracked and modified with the `.timer`, `.timer_lap` and `.timer_total` variables within `cat_magic()` or `message_magic()`.

Note that the timer is precise at +/- 1ms, hence it should **not** be used to time algorithms with very short execution times.

It works by saving the current system time in R options (`stringmagic_timer` and `stringmagic_timer_origin`). Hence, since it uses options, it should not be used in parallel processes.

Value

This function does not return anything and is only intended to be used in conjunction with future calls of `string_magic()`.

Author(s)

Laurent Berge

See Also

Other tools with aliases: `string_clean_alias()`, `string_magic()`, `string_magic_alias()`, `string_ops_alias()`, `string_vec_alias()`

Examples

```
# simple example where we time the execution of some elements in a function
# we trigger the message conditionally on the value of the argument `debug`.
rnorm_crossprod = function(n, mean = 0, sd = 1, debug = FALSE){
  # we set the timer
  timer_magic()
  # we compute some stuff
  x = rnorm(n, mean, sd)
  # we can report the time with .timer
  message_magic("{15 align ! Generation}: {.timer}", .trigger = debug)

  res = x %*% x
  message_magic("{15 align ! Product}: {.timer}",
                "{15 align ! Total}: {.timer_total}",
                .sep = "\n", .trigger = debug)
  res
}

rnorm_crossprod(1e5, TRUE)
```

Index