Package 'tidyoperators'

March 19, 2023

Title Infix operators for tidier R code

Version 0.0.9

Description The 'tidyoperators' R-package adds some much needed infix operators,
and a few functions,
to make your R code much more tidy.
It includes infix operators for the negation of logical operators
(exclusive-or, not-and, not-in),
safer float (in)equality operators,
in-place modifying mathematical arithmetic,
string arithmetic,
string sub-setting,
in-place modifying string arithmetic,
in-place modifying string sub-setting,
in-place modifying unreal replacers,
and infix operators for custom row- and column-wise rank-based ordering of matrices.
The 'tidyoperators' R-package also adds the stringi-like
stri_locate_ith and stri_join_mat functions.
It also adds string functions to replace, extract, add-on, transform, and re-arrange,
the ith pattern occurrence or position.
Most stringi pattern expressions options
are available for the string-pattern-related functions, when appropriate.
This package adds the transform_if function.
This package also allows integrating third-party parallel computing packages
(like stringfish)
for some of its functions.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests knitr,
rmarkdown,
stringfish (>= 0.15.7),
testthat (>= 3.0.0)

VignetteBuilder knitr

Depends R (>= 4.1.0)

Imports stringi (>= 1.7.12)

R topics documented:

`float_logic` *Safer float (in)equality operators*

Description

The `%f==%`, `%f!=%` `%f<%`, `%f>%`, `%f<=%`, `%f>=%` operators perform "float logic". They are virtually equivalent to the regular (in)equality operators,
`==`, `!=`, `<`, `>`, `<=`, `>=`,
except for one aspect. The float logic operators assume that if the absolute difference between `x` and `y` is smaller than the Machine tolerance, `sqrt(.Machine$double.eps)`, then `x` and `y` ought to be consider to be equal.
Thus these provide safer float logic.
For example: `0.1*7 == 0.7` returns `FALSE`, even though they are equal, due to the way floating numbers are stored in programming languages like R. But `0.1*7 %f==% 0.7` returns `TRUE`.

Usage

```
x %f==% y

x %f!=% y

x %f<% y

x %f>% y

x %f<=% y

x %f>=% y
```

Arguments

x, y	numeric vectors, matrices, or arrays, though these operators were specifically designed for floats (class "double").

Examples

```
x <- c(0.3, 0.6, 0.7)
y <- c(0.1*3, 0.1*6, 0.1*7)
print(x); print(y)
x == y # gives FALSE, but should be TRUE
x!= y # gives TRUE, should be FALSE
x > y # not wrong
x < y # gives TRUE, should be FALSE
x %f==% y # here it's done correctly
x %f!=% y # correct
x %f<% y # correct
x %f>% y # correct
x %f<=% y # correct
x %f>=% y # correct

# These operators still work for non-float numerics also:
x <- 1:5
y <- 1:5
x %f==% y
x %f!=% y
x %f<% y
x %f>% y
x %f<=% y
x %f>=% y

x <- 1:5
y <- x+1
x %f==% y
x %f!=% y
x %f<% y
x %f>% y
x %f<=% y
x %f>=% y

x <- 1:5
y <- x-1
x %f==% y
x %f!=% y
x %f<% y
x %f>% y
x %f<=% y
x %f>=% y
```

---

inplace_math *In-place modifying mathematical arithmetic*

Description

In-place modifiers for addition, subtraction, multiplication, division, power, root, logarithm, and anti-logarithm.

`x %+ <-% y` is the same as `x <- x + y`

`x %- <-% y` is the same as `x <- x - y`

`x %* <-% y` is the same as `x <- x * y`

`x %/ <-% y` is the same as `x <- x / y`

`x %^ <-% p` is the same as `x <- x^p`

`x %rt <-% p` is the same as `x <- x^(1/p)`

`x %logb <-% b` is the same as `x <- log(x, base=b)`

`x %alogb <-% b` is the same as `x <- b^x`; if `b=exp(1)`, this is the same as `x <- exp(x)`

Usage

```
x %+ <-% y

x %- <-% y

x %* <-% y

x %/ <-% y

x %^ <-% p

x %rt <-% p

x %logb <-% b

x %alogb <-% b
```

Arguments

x	a number or numeric (or 'number-like') vector, matrix, or array.
y	a number, or numeric (or 'number-like') vector, matrix, or array of the same length/dimension as `x`. It gives the number to add, subtract, multiply by, or divide by.
p	a number, or a numeric vector of the same length as `x`. It gives the power to be used.
b	a number, or a numeric vector of the same length as `x`. It gives the logarithmic base to be used.

Value

These operators do not return any value: they are in-place modifiers, and thus modify x directly.

Examples

```
x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %+ <-% 3 # same as x <- x + 3
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %- <-% 3 # same as x <- x - 3
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %* <-% 3 # same as x <- x * 3
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %/ <-% 3 # same as x <- x / 3
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %^ <-% 3 # same as x <- x^3
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %rt <-% 3 # same as x <- x^(1/3)
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %logb <-% 3 # same as x <- log(x, base=3)
print(x)

x <- matrix(rpois(10, 10), ncol=2)
print(x)
x %alogb <-% 3 # same as x <- 3^x
print(x)

x <- 3
print(x)
x %alogb <-% exp(1) # same as x <- exp(x)
print(x)
exp(3) # notice this is the same.
```

`inplace_str_arithmetic`

In place modifying string arithmetic

Description

In-place modifier versions of string arithmetic:

`x %s+ <-% y` is the same as `x <- x %s+% y`

`x %s- <-% p` is the same as `x <- x %s-% p`

`x %s* <-% n` is the same as `x <- x %s*% n`

`x %s/ <-% p` is the same as `x <- x %s/% p`

See also the documentation on string arithmetic: string arithmetic.
Note that there is no in-place modifier versions of `%ss%`, `s_extract()`, and `s_repl()`.

Usage

```
x %s+ <-% y

x %s- <-% p

x %s* <-% n

x %s/ <-% p
```

Arguments

`x, y, p, n`	see string arithmetic and s_pattern.

Value

These operators do not return any value: they are in-place modifiers, and thus modify `x` directly.

Examples

```
y <- "a"
p <- "a|e|i|o|u"
n <- c(2, 3)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s+ <-% y # same as x <- x %s+% y
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s- <-% p # same as x <- x %s-% p
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s* <-% n # same as x <- x %s\*% n
```

```
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s/ <-% p # same as x <- x %s/% p
print(x)


################################################################################


y <- "a"
# pattern with ignore.case=TRUE:
p <- s_pattern(regex = "A|E|I|O|U", ignore.case=TRUE)
n <- c(3, 2)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s+ <-% y # same as x <- x %s+% y
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s- <-% p # same as x <- x %s-% p
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s* <-% n # same as x <- x %s\*% n
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s/ <-% p # same as x <- x %s/% p
print(x)
```

`inplace_str_subset` *In place modifying string subsetting*

Description

In-place modifier versions of string subsetting:

`x %sget <-% ss` is the same as `x <- x %sget% ss`

`x %strim <-% ss` is the same as `x <- x %strim% ss`

See also the documentation on string subsetting (string subset).
Note that there is no in-place modifier versions of `%ss%`.

Usage

```
x %sget <-% ss

x %strim <-% ss
```

Arguments

x, ss	see string subset.

Value

These operators do not return any value: they are in-place modifiers, and thus modify x directly.

Examples

```
ss <- c(2,2)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %sget <-% ss # same as x <- x %sget% ss
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %strim <-% ss # same as x <- x %strim% ss
print(x)


################################################################################


ss <- c(2,2)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %sget <-% ss # same as x <- x %sget% ss
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %strim <-% ss # same as x <- x %strim% ss
print(x)
```

`inplace_unreal` *In-place unreal replacers*

Description

In-place modifiers to replace unreal (NA, NaN, Inf, -Inf) elements.
Works on vectors, matrices, and arrays.
The following
`x %unreal <-% 0`

is the same as
`x[is.na(x)|is.nan(x)|is.infinite(x)] <- 0`

Usage

```
x %unreal <-% replacement
```

Arguments

`x`	a vector or matrix whose unreal values are to be replaced.
`replacement`	the replacement value.

Value

This operator does not return any value: it is an in-place modifiers, and thus modifies x directly. The x vector is modified such that all NA, NaN and infinities are replaced with the given replacement value.

Examples

```
x <- c(1:9, NA, NaN, Inf)
print(x)
x %unreal <-% 0 # same as x[is.na(x)|is.nan(x)|is.infinite(x)] <- 0
print(x)
```

---

logic_ops *Logic operators*

Description

Additional logic operators:

The `x %xor% y` operator is the "exclusive-or" operator, the same as `xor(x, y)`.

The `x %n&% ` operator is the "not-and" operator, the same as `(!x) & (!y)`.

The `x %out% y` operator is the same as `!x %in% y`.

The `x %?=% y` operator checks if `x` and `y` are **both** unreal or unknown (i.e. NA, NaN, Inf, -Inf).

The `n %=numtype% numtype` operator is a vectorized operator that checks for every value of numeric vector `n` if it can be considered a number belonging to type `numtype`. See arguments for details.

The `s %=strtype% strtype` operator is a vectorized operator that checks for every value of character vector `s`
if it can seen as a certain `strtype`. See arguments for details.

The `s %sgrep% p` operator is a vectorized operator that checks for every value of character vector `s` if it has pattern `p`.

Usage

```
x %xor% y

x %n&% y

x %out% y

x %?=% y

s %sgrep% p

n %=numtype% numtype

s %=strtype% strtype
```

Arguments

x, y	see Logic.
s	a character vector.
p	the result from s_pattern, or else a character vector of the same length as s with regular expressions.
n	a numeric vector.
numtype	a single string giving the type if numeric to be checked. The following options are supported:

- "~0": zero, or else a number whose absolute value is smaller than the Machine tolerance (sqrt(.Machine$double.eps)).
- "B": binary numbers (exactly 0 or exactly 1);
- "prop": proportions - numbers between 0 and 1 (exactly 0 or 1 is also allowed);
- "N": Natural numbers (non-negative integers including zero);
- "I": Integers;
- "odd": odd integers;
- "even": even integers;
- "R": Real numbers;
- "unreal": infinity, NA, or NaN;

strtype	a single string giving the type of string to be checked. The following options are supported:

- "empty": checks if the string only consists of empty spaces.
- "unreal": checks if the string is NA, or if it has literal string "NA", "NaN" or "Inf", regardless if it has leading or trailing spaces.
- "numeric": checks if the string can be converted to a number, disregarding leading and trailing spaces. I.e. the string "5.0" can be converted to the the actual number 5.0.
- "special": checks if the string consists of only special characters.

Examples

```
x <- c(TRUE, FALSE, TRUE, FALSE, NA, FALSE, TRUE)
y <- c(FALSE, TRUE, TRUE, FALSE, NA, NA, NA)
cbind(x, y, "x %xor% y"=x %xor% y, "x %n&% y" = x %n&% y, "x %?=% y" = x %?=% y)

1:3 %out% 1:10
1:10 %out% 1:3


n <- c(0:5, 0:-5, 0.1, -0.1, 0, 1, Inf, -Inf, NA, NaN)
1e-20 %=numtype% "~0"
n[n %=numtype% "B"]
n[n %=numtype% "prop"]
n[n %=numtype% "N"]
n[n %=numtype% "I"]
n[n %=numtype% "odd"]
n[n %=numtype% "even"]
n[n %=numtype% "R"]
n[n %=numtype% "unreal"]

s <- c(" AbcZ123 ", " abc ", " 1.3 ", " !#$%^&*() ", "  ", "  NA  ", "  NaN  ", " Inf ")
s[s %=strtype% "empty"]
s[s %=strtype% "unreal"]
s[s %=strtype% "numeric"]
s[s %=strtype% "special"]


s <- c("Hello world", "Goodbye world")
p <- s_pattern(regex = c("Hello", "Hello"))
s %sgrep% p
```

---

matrix_ops *Matrix custom re-ordering operators*

Description

Infix operators for custom row- and column-wise rank-based re-ordering of matrices

The x %row~% rank operator re-orders the elements of every row of matrix x according to the rank

given in matrix rank.

The x %col~% rank operator re-orders the elements of every column of matrix x according to the rank given in matrix rank.

Usage

```
x %row~% rank

x %col~% rank
```

Arguments

x	a matrix
rank	a matrix with the same dimensions as x, giving the ordering rank of every element of matrix x.

Details

If matrix x is a numeric matrix, and one wants to sort the elements of every row or column numerically, x %row~% x or x %col~% x would suffice, respectively.

If matrix x is not numeric, x %row~% x and x %col~% x are still possible, but probably not the best option. In the non-numeric case, providing a ranking matrix for rank would be faster and give more accurate ordering. See the examples section.

These operators internally only use vectorized operations (no loops or apply-like functions), and are faster than re-ordering matrices using loops or apply-like functions.

Value

A modified matrix.

Examples

```
# numeric matrix ====

mat <- matrix(sample(1:25), nrow=5)
print(mat)
mat %row~% mat # sort elements of every row
mat %row~% -mat # reverse-sort elements of every row
mat %col~% mat # sort elements of every column
mat %col~% -mat # reverse-sort elements of every column

mat <- matrix(sample(1:25), nrow=5)
print(mat)
rank <- sample(1:length(mat)) |> matrix(ncol=ncol(mat)) # randomized rank
mat %row~%  rank# randomly shuffle every row independently
mat %col~% rank # randomize shuffle every column independently

# character matrix ====
```

```
mat <- matrix(sample(letters, 25), nrow=5)
print(mat)
rank <- stringi::stri_rank(as.vector(mat)) # alphabetic ranking from stringi
rank <- matrix(rank, ncol=ncol(mat)) # rank matrix
mat %row~% rank # sort elements of every row
mat %row~% -rank # reverse-sort elements of every row
mat %col~% rank # sort elements of every column
mat %col~% -rank # reverse-sort elements of every column

mat <- matrix(sample(letters, 25), nrow=5)
print(mat)
rank <- sample(1:length(mat)) |> matrix(ncol=ncol(mat)) # randomized rank
mat %row~%  rank# randomly shuffle every row independently
mat %col~% rank # randomize shuffle every column independently
```

`stri_join_mat` *Concatenate Character Matrix Row-wise or Column-wise*

Description

The `stri_join_mat()` function (and their aliases `stri_c_mat` and `stri_paste_mat`) perform row-wise (`margin=1`; the default) or column-wise (`margin=2`) joining of a matrix of strings, thereby transforming a matrix of strings into a vector of strings.

Usage

```
stri_join_mat(mat, margin = 1, sep = "", collapse = NULL)

stri_c_mat(mat, margin = 1, sep = "", collapse = NULL)

stri_paste_mat(mat, margin = 1, sep = "", collapse = NULL)
```

Arguments

`mat`	a matrix of strings
`margin`	the margin over which the strings must be joined. If `margin=1`, the elements in each row of matrix `mat` are joined into a single string. Thus if the matrix has 10 rows, it returns a vector of 10 strings. If `margin=2`, the elements in each column of matrix `mat` are joined into a single string. Thus if the matrix has 10 columns, it returns a vector of 10 strings.
`sep, collapse`	as in stri_join.

Details

The examples section show the uses of the `stri_join_mat()` function.

Value

The stri_join_mat() function, and its aliases, return a vector of strings.

Examples

```
# re-ordering characters in strings ====

x <- c("Hello world", "Goodbye world")
print(x)
mat <- stringi::stri_split_boundaries(x, simplify = TRUE, type="character")
rank <- stringi::stri_rank(as.vector(mat)) |>  matrix(ncol=ncol(mat))
sorted <- mat %row~% rank
print(sorted)
stri_join_mat(sorted, margin=1)
stri_join_mat(sorted, margin=2)

##############################################################################

# re-ordering words ====

x <- c("Hello everyone", "Goodbye everyone")
print(x)
mat <- stringi::stri_split_boundaries(x, simplify = TRUE, type="word")
rank <- matrix(stringi::stri_rank(as.vector(mat)), ncol=ncol(mat))
sorted <- mat %row~% rank
print(sorted)
stri_c_mat(sorted, margin=1) # <- alias for stri_join_mat
stri_c_mat(sorted, margin=2)

##############################################################################

# re-ordering sentences ====

x <- c("Hello, who are you? Oh, really?! Cool!", "I don't care. But I really don't.")
print(x)
mat <- stringi::stri_split_boundaries(x, simplify = TRUE, type="sentence")
rank <- matrix(stringi::stri_rank(as.vector(mat)), ncol=ncol(mat))
sorted <- mat %row~% rank
print(sorted)
stri_paste_mat(sorted, margin=1) # <- another alias for stri_join_mat
stri_paste_mat(sorted, margin=2)
```

---

stri_locate_ith *Locate iˆth Pattern Occurrence*

Description

The stri_locate_ith function locates the i^{th} occurrence of a pattern in each string of some character vector.

Usage

```
stri_locate_ith(str, i, ..., regex, fixed, coll, charclass, simplify = FALSE)
```

Arguments

`str`	a string or character vector.
`i`	a number, or a numeric vector of the same length as `str`. This gives the i^{th} instance to be replaced. Positive numbers are counting from the left. Negative numbers are counting from the right. I.e.: `stri_locate_ith(str, i=1, p, rp)` gives the position (range) of the first occurrence of pattern `p`. `stri_locate_ith(str, i=-1, p, rp)` gives the position (range) of the last occurrence of pattern `p`. `stri_locate_ith(str, i=2, p, rp)` gives the position (range) of the second occurrence of pattern `p`. `stri_locate_ith(str, i=-2, p, rp)` gives the position (range) of the second-last occurrence of pattern `p`. If `abs(i)` is larger than the number of instances, the first (if `i < 0`) or last (if `i > 0`) instance will be given. For example: suppose a string has 3 instances of p; then if `i = 4` the third instance will be located, and if `i = -4` the first instance will be located.
`...`	more arguments to be supplied to stri_locate.
`regex, fixed, coll, charclass`	character vector of search patterns, as in stri_locate.
`simplify`	either `TRUE` or `FALSE` (default = `FALSE`): • If `FALSE`, `stri_locate_ith` returns a list, usable in the stri_sub_all functions (for example: to transform all matches). • If `TRUE`, `stri_locate_ith` returns an integer matrix of positions and lengths.

Value

If `simplify = FALSE`, `stri_locate_ith` returns a list, one element for each string. Each list element consists of a matrix with 2 columns and one row:
The first column gives the start position of the i^{th} occurrence of pattern `p`.
The second column gives the end position of the i^{th} occurrence of pattern `p`.
When `simplify=FALSE`, the results can be used in the `from` argument in the stri_sub_all functions, for example to transform the i^{th} matches (see examples section below).

If `simplify = TRUE` (default), `stri_locate_ith` this returns an integer matrix with 3 columns:
The first column gives the start position of the i^{th} occurrence of pattern `p`.
The second column gives the end position of the i^{th} occurrence of pattern `p`.
The third column gives the length of the position range of the i^{th} occurrence of pattern `p`.

Examples

```
# simple pattern ====
```

```
x <- rep(paste0(1:10, collapse=""), 10)
print(x)
out <- stri_locate_ith(x, 1:10, regex = as.character(1:10), simplify=TRUE)
cbind(1:10, out)


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- rep("a|e|i|o|u",2)
out <- stri_locate_ith(x, c(-1, 1), regex=p, simplify=TRUE)
print(out)
substr(x, out[,1], out[,2])


#############################################################################

# ignore case pattern ====


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- rep("A|E|I|O|U", 2)
out <- stri_locate_ith(x, c(1, -1), regex=p, case_insensitive=TRUE, simplify=TRUE)
substr(x, out[,1], out[,2])


#############################################################################

# multi-character pattern ====

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
# multi-character pattern:
p <- rep("AB", 2)
out <- stri_locate_ith(x, c(1, -1), regex=p, simplify=TRUE, case_insensitive=TRUE)
print(out)
substr(x, out[,1], out[,2])



#############################################################################

# Replacement transformation using stringi ====

x <- c("hello world", "goodbye world")
loc <- stri_locate_ith(x, c(1, -1), regex="a|e|i|o|u", simplify = FALSE)
extr <- stringi::stri_sub_all(x, from=loc)
repl <- lapply(extr, \(x)chartr(x, old = "a-zA-Z", new = "A-Za-z"))
stringi::stri_sub_all_replace(x, loc, replacement=repl)
```

`str_arithmetic` *String arithmetic*

Description

String arithmetic operators.

The `x %s+% y` operator is equivalent to `paste0(x,y)`.

The `x %s-% p` operator removes character/pattern defined in `p` from `x`.

The `x %s*% n` operator repeats every element of `x` for `n` times, and glues them together.

The `x %s/% p` operator counts how often regular expression or character pattern `p` occurs in each element of `x`.

Usage

```
x %s+% y

x %s-% p

x %s*% n

x %s/% p
```

Arguments

x	a string or character vector.
y	a string, or a character vector of the same length as x.
p	the result from s_pattern, or else a character vector of the same length as x with regular expressions.
n	a number, or a numeric vector of the same length as x.

Details

These operators and functions serve as a way to provide straight-forward string arithmetic, missing from base R.

Value

The `%s+%`, `%s-%`, and `%s*%` operators return a character vector of the same length as `x`.
The `%s/%` returns a integer vector of the same length as `x`.

Examples

```
x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
y <- c("a", "b")
p <- rep("a|e|i|o|u", 2) # same as p <- s_pattern(regex=rep("a|e|i|o|u", 2))
n <- c(3, 2)

x %s+% y # =paste0(x,y)
```

```
x %s-% p # remove all vowels from x
x %s*% n
x %s/% p # count how often vowels appear in each string of vector x.


##############################################################################


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
y <- "a"
# pattern that ignores case:
p <- s_pattern(regex=rep("A|E|I|O|U", 2), ignore.case=TRUE)
n <- c(2, 3)

x %s+% y # =paste0(x,y)
x %s-% p # remove all vowels from x
x %s*% n
x %s/% p # count how often vowels appears in each string of vector x.
```

`str_subset_ops` *String subsetting operators*

Description

String subsetting operators.

The `x %ss% s` operator allows indexing a single string as-if it is an iterable object.

The `x %sget% ss` operator gives a certain number of the first and last characters of `x`.

The `x %strim% ss` operator removes a certain number of the first and last characters of `x`.

Usage

```
x %ss% s

x %sget% ss

x %strim% ss
```

Arguments

`x`	a string or character vector.
`s`	a numeric vector giving the subset indices.
`ss`	a vector of length 2, or a matrix with 2 columns with `nrow(ss)==length(x)`. The object `ss` should consist entirely of non-negative integers (thus 0, 1, 2, etc. are valid, but -1, -2, -3 etc are not valid). The first element/column of `ss` gives the number of characters counting from the left side to be extracted/removed from `x`. The second element/column of `ss` gives the number of characters counting from the right side to be extracted/removed from `x`.

Details

These operators serve as a way to provide straight-forward string sub-setting.

Value

The `%ss%` operator always returns a vector or matrix, where each element is a single character.

Examples

```
x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
ss <- c(2,3)
x %sget% ss

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
ss <- c(1,0)
x %sget% ss

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
ss <- c(2,3)
x %strim% ss

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
ss <- c(1,0)
x %strim% ss
```

`substr_repl` *Sub-string functions*

Description

Fully vectorized sub-string functions.
These functions extract, replace, add-in, transform, or re-arrange, the i^{th} pattern occurrence or position range.

The `substr_repl(x, rp, ...)` function replaces a position (range) with string `rp`.

The `substr_chartr(x, old, new, ...)` function transforms the sub-string at a position (range) using `chartr(old, new)`.

The `substr_addin(x, addition, side, ...)` function adds the additional string `addition` at the side (specified by argument `side`) of a position.

The `substr_extract(x, type, ...)` function extracts the string at, before, or after some position.

The `substr_arrange(x, arr, ...)` function sorts (alphabetically or reverse-alphabetically) or reverse the sub-string at a position (range).

Usage

```
substr_repl(x, rp, ..., loc = NULL, start = NULL, end = NULL, fish = FALSE)

substr_chartr(
  x,
  old = "a-zA-Z",
  new = "A-Za-z",
  ...,
  loc = NULL,
  start = NULL,
  end = NULL,
  fish = FALSE
)

substr_addin(
  x,
  addition,
  side = "after",
  ...,
  loc = NULL,
  at = NULL,
  fish = FALSE
)

substr_extract(
  x,
  type = "at",
  ...,
  loc = NULL,
  start = NULL,
  end = NULL,
  fish = FALSE
)

substr_arrange(
  x,
  arr = "incr",
```

```
  ...,
  loc = NULL,
  start = NULL,
  end = NULL,
  opts_collator = NULL,
  fish = FALSE
)
```

Arguments

x	a string or character vector.
rp	a string, or a character vector of the same length as x, giving the replacing strings.
...	only applicable if fish=TRUE; other arguments to be passed to the stringfish functions.
loc	The result from the stri_locate_ith function. It does not matter if the result is in the list form (simplify = FALSE), or in the matrix form (simplify = TRUE). See stri_locate_ith. NOTE: you cannot fill in both loc and start,end, or both loc and at. Choose one or the other.
start, end	integers, or integer vectors of the same length as x, giving the start and end position of the range to be modified.
fish	although tidyoperators has no dependencies other than stringi, it does allow the internal functions to use the multi-threadable stringfish functions. To do so, set fish=TRUE; this requires stringfish to be installed.
old, new	see chartr. Defaults to old="a-zA-Z", new="A-Za-z", which means upper case characters will be transformed to lower case characters, and vice-versa.
addition	a string, or a character vector of the same length as x, giving the string(s) to add-in.
side	which side of the position to add in the string. Either "before" or "after".
at	an integer, or integer vector of the same length as x.
type	a single string, giving the part of the string to extract. 3 options available: • type = "at": extracts the string part at the position range; • type = "before": extracts the string part before the position range; • type = "after": extracts the string part after the position range.
arr	a single string, giving how the sub-string should be arranged. 3 options available: • arr = "incr": sort the sub-string alphabetically. • arr = "decr": sort the sub-string reverse alphabetically. • arr = "rev": reverse the sub-string. • arr = "rand": randomly shuffles the sub-string.
opts_collator	as in stri_rank. Only used when arr = "incr" or arr = "decr".

Details

These functions serve as a way to provide straight-forward sub-string modification and/or extraction.
All substr_ functions internally only use fully vectorized R functions (no loops or apply-like functions).

Value

A modified character vector. If no match is found in a certain string of character vector `x`, the unmodified string is returned. The exception is for the `substr_extract()` function: in this function, non-matches return `NA`.

Examples

```
# numerical substr ====

x <- rep("12345678910", 2)
start=c(1, 2); end=c(3,4)
substr_extract(x, start=start, end=end)
substr_extract(x, type="before", start=start, end=end)
substr_extract(x, type="after", start=start, end=end)
substr_repl(x, c("??", "!!"), start=start, end=end)
substr_chartr(x, start=start, end=end)
substr_addin(x, c(" ", "~"), "after", at=end)
substr_addin(x, c(" ", "~"), "before", at=start)
substr_arrange(x, start=start, end=end)
substr_arrange(x, "decr", start=start, end=end)
substr_arrange(x, "rev", start=start, end=end)
substr_arrange(x, "rand", start=start, end=end)

start=10; end=11
substr_extract(x, start=start, end=end)
substr_extract(x, type="before", start=start, end=end)
substr_extract(x, type="after", start=start, end=end)
substr_repl(x, "??", start=start, end=end)
substr_chartr(x, start=start, end=end)
substr_addin(x, " ", "after", at=end)
substr_addin(x, " ", "before", at=start)

start=5; end=6
substr_extract(x, start=start, end=end)
substr_extract(x, type="before", start=start, end=end)
substr_extract(x, type="after", start=start, end=end)
substr_repl(x, "??", start=start, end=end)
substr_chartr(x, start=start, end=end)
substr_addin(x, " ", "after", at=end)
substr_addin(x, " ", "before", at=start)


################################################################################

# simple pattern ====

x <- c("goodGOODGoodgOOd", "goodGOODGoodgOOd", paste0(letters[1:13], collapse=""))
print(x)
```

```
loc <- stri_locate_ith(
  # locate second-last occurrence of "good" of each string in x:
  x, -2, regex="good", case_insensitive=TRUE
)
substr_extract(x, loc=loc) # extract second-last "good"
substr_repl(x, c("??", "!!", " "), loc=loc) # replace second-last "good"
substr_chartr(x, loc=loc) # switch upper/lower case of second-last "good"
substr_addin(x, c(" ", "~", " "), "after", loc=loc) # add white space after second-last "good"
substr_addin(x, c(" ", "~", " "), "before", loc=loc) # add white space before second-last "good"
substr_arrange(x, loc=loc) # sort second-last "good"
substr_arrange(x, "decr", loc=loc) # reverse-sort second-last "good"
substr_arrange(x, "rev", loc=loc) # reverse second-last "good"
substr_arrange(x, "rand", loc=loc) # randomly shuffles "good"
```

`s_pattern` *Pattern attribute assignment*

Description

The `%s-%` and `%s/%` operators, their in-place equivalents, as well as the `%sgrep%` operator, all perform pattern matching for some purpose. By default the pattern matching is interpreted as case-sensitive `regex` patterns from `stringi`.

The `s_pattern` function allows the user to specify exactly how the pattern should be interpreted. To use more refined pattern definition, simply replace the right-hand-side expression `p` in the relevant operators with a call from the `s_pattern()` function.

The `s_pattern()` function uses the exact same argument convention as `stringi`. For example:

- `s_pattern(regex=p, case_insensitive=FALSE, ...)`
- `s_pattern(fixed=p, ...)`
- `s_pattern(coll=p, ...)`
- `s_pattern(boundary=p, ...)`
- `s_pattern(charclass=p, ...)`

All arguments in `s_pattern()` are simply passed to the appropriate functions in `stringi`.
For example:
`x %s/% p` counts how often regular expression `p` occurs in `x`,
whereas `x %s/% s_pattern(fixed=p, case_insensitive=TRUE)` will do the same, except it uses fixed (i.e. literal) expression, and it does not distinguish between upper case and lower case characters.
For consistency with base R and with packages such as `stringr`, one can also fill in `ignore.case=TRUE` or `ignore_case=TRUE` instead of `case_insensitive=TRUE`, and `s_pattern` will still understand that.

Usage

```
s_pattern(...)
```

Arguments

...	pass stringi arguments here. I.e. regex=p, boundary=p, coll=p, charclass=p, case_insensitive=FALSE, etc. See the documentation in the stringi R package.

Details

The s_pattern() function only works in combination with the functions and operators in this package. It does not affect functions from base R or functions from other packages.

Value

The s_pattern(...) call returns a list with arguments that will be passed to the appropriate functions in stringi.

Examples

```
x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- rep("a|e|i|o|u", 2) # same as p <- s_pattern(regex=rep("a|e|i|o|u", 2))
x %s/% p # count how often vowels appear in each string of vector x.

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- s_pattern(regex=rep("A|E|I|O|U", 2), ignore.case=TRUE)
x %s/% p # count how often vowels appear in each string of vector x.
```

---

tidyoperators_help *The tidyoperators help page*

Description

Welcome to the tidyoperators help page!

The 'tidyoperators' R-package adds some much needed infix operators, and a few functions, to make your R code much more tidy. It includes infix operators for the negation of logical operators (exclusive-or, not-and, not-in), safer float (in)equality operators, in-place modifying mathematical arithmetic, string arithmetic, string sub-setting, in-place modifying string arithmetic, in-place modifying string sub-setting, in-place modifying unreal replacers, and infix operators for custom row- and column-wise rank-based ordering of matrices. The 'tidyoperators' R-package also adds the stringi-like stri_locate_ith and stri_join_mat functions. It also adds string functions to replace, extract, add-on, transform, and re-arrange, the ith pattern occurrence or position. And it includes some helper functions for more complex string arithmetic. Most stringi pattern expressions options are available for the string-pattern-related functions, when appropriate. This package adds the transform_if function. This package also allows integrating third-party parallel computing packages (like stringfish) for some of its functions.

The `tidyoperators` R package adds the following functionality:

- Infix logical operators for exclusive-or, not-and, not-in, number-type, and string-type.
- Safer (in)equality operators for floating numbers.
- Infix operators for In-place modifiers for mathematical arithmetic.
- Infix operators for string arithmetic.
- Infix operators for string sub-setting.
- Infix operators for In-place modifying string arithmetic.
- Infix operators for In-place modifying string sub-setting.
- The in-place modifying unreal replacer operator.
- Infix operators for custom row- and column-wise re-ordering of matrices.
- The `tidyoperators` package adds additional `stringi` functions, namely stri_locate_ith and stri_join_mat (and aliases). These functions use the same naming and argument convention as the rest of the `stringi` functions, thus keeping your code consistent.
- The fully vectorized sub-string functions, that extract, replace, add-in, transform, or re-arrange, the ith pattern occurrence or location.
- The s_pattern helper function for string operators.
- The transform_if function, and some related infix operators.
- Most `stringi` pattern expressions options are available for the string-pattern-related functions, when appropriate.
- This R package has only one dependency: `stringi`. No other dependencies, as to avoid "dependency hell".
- Although this package has no other dependencies, it allows multi-threading of functions (when appropriate) through third-party packages (like `stringfish`).

Please also have a look at the Read-Me file on the Github main page of this package: https://github.com/tony-aw/tidyoperators

Usage

```
tidyoperators_help()
```

transform_if *The transform_if function and the subset_if operators*

Description

Consider the following code:
`x[cond(x)] <- f(x[cond(x)])`
Here a conditional subset of the object `x` is transformed with function `f`, where the condition is using a function referring to `x` itself (namely `cond(x)`). Consequently, reference to `x` is written four times!
The `tidyoperators` package therefore adds the `transform_if()` function which will tidy this up.

`x <- transform(x, cond, trans)`
is exactly equivalent to
`x[cond(x)] <- trans(x[cond(x)])`

Besides `transform_if`, the `tidyoperators` package also adds 2 "subset_if" operators:

The `x %[if]% cond` operator selects elements from vector/matrix/array `x`, for which the result of `cond(x)` returns `TRUE`.

The `x %[!if]% cond` operator selects elements from vector/matrix/array `x`, for which the result of `cond(x)` returns `FALSE`.

Usage

```
transform_if(x, cond, trans = NULL)

x %[if]% cond

x %[!if]% cond
```

Arguments

`x`	a vector, matrix, or array.
`cond`	a function that returns a binary logic (`TRUE`, `FALSE`) vector of the same length/dimensions as `x` (for example: `is.na`). • Elements of `x` for which `cond(x)==TRUE` are transformed / selected; • Elements of `x` for which `cond(x)==FALSE` are not transformed /selected.
`trans`	the transformation function to use. For example: `log`.

Details

The `transform_if(x, cond, trans)` function does not rely on any explicit or implicit loops, nor any third-party functions.

Value

The `transform_if()` function returns the same object `x`, with the same dimensions, except with the subset transformed.
Note that this function **returns** object `x`, to modify `x` directly, one still has to assign it. To keep your code tidy, consider combining this function with `magrittr`'s in-place modifying piper-operator (`%<>%`). I.e.:
`very_long_name_1 %<>% transform_if(cond, trans)`

The subset_if - operators all return a vector with the selected elements.

Examples

```
object_with_very_long_name <- matrix(-10:9, ncol=2)
print(object_with_very_long_name)
object_with_very_long_name |> transform_if(\(x)x>0, log)
object_with_very_long_name %[if]% \(x)x %in% 1:10
object_with_very_long_name %[!if]% \(x)x %in% 1:10
```

`%m import <-%` *Utility operator*

Description

The `alias %m import <-% pkgs` operator imports multiple R package under the same alias.

The `alias %m import <-% pkgs` command is essentially the same as
`alias <- loadNamespace("packagename")`
except the `alias %m import <-% pkgs` operator allows assigning multiple packages to the same alias, and this operator does not import internal functions (i.e. internal functions are kept internal, as they should).

For example:
`fv %m import <-% c("data.table", "collapse", "tidytable")`

The `alias %m import <-% pkgs` operator will tell the user about conflicting objects. It will also inform the user when importing a package that consists mostly of infix operators.

Usage

```
alias %m import <-% pkgs
```

Arguments

`alias`	a variable name (unquoted), giving the (not yet existing) object where the package(s) are to be assigned to.
`pkgs`	a character vector with the package name(s). NOTE: The order matters! If 2 packages share objects with the same name, the package named last will overwrite the earlier named package.

Value

The variable named in the `alias` argument will be created (if it did not already exist), and it will contain the (merged) package environment.

Examples

```
## Not run:
fv %m import <-% c("data.table", "collapse", "tidytable")

## End(Not run)
```

Index