

Package ‘tidyoperators’

June 4, 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 additional logic operators, safer float (in)equality operators, and infix operators for custom row- and column-wise ordering of matrices. It also adds some stringi-based string related functions and operators. It also adds operators and a few functions to help reduce unnecessary repetitive code. And finally, it also adds some functions and an operator for easier package/library management. The 'tidyoperators' R-package has only one dependency, namely 'stringi', though it does allow multi-threading of some of the string-related functions (when appropriate) via the suggested 'stringfish' R-package.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests this.path (>= 1.3.0),
rlang,
knitr,
rmarkdown,
stringfish (>= 0.15.7),
testthat (>= 3.0.0)

VignetteBuilder knitr

Depends R (>= 4.1.0)

Imports stringi (>= 1.7.12)

Config/testthat/edition 3

URL <https://github.com/tony-aw/tidyoperators>

BugReports <https://github.com/tony-aw/tidyoperators/issues>

R topics documented:

float_logic	2
force_libPaths	4

import	5
inplace_math	7
inplace_str_arithmetic	9
inplace_str_subset	11
logic_ops	12
matrix_ops	15
stri_join_mat	16
stri_locate_ith	18
str_arithmetic	20
str_subset_ops	21
substr_repl	23
s_pattern	26
tidyoperators_help	27
transform_if	29

Index	31
--------------	-----------

float_logic	<i>Safer float (in)equality operators</i>
-------------	---

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.

There are also the `x %f{ }% bnd` and `x %f!{ }% bnd` operators, where `bnd` is a vector of length 2, or a 2-column matrix (`nrow(bnd)==length(x)` or `nrow(bnd)==1`).

The `x %f{ }% bnd` operator checks if x is within the closed interval with bounds defined by `bnd`.

The `x %f!{ }% bnd` operator checks if x is outside the closed interval with bounds defined by `bnd`.

Usage

`x %f==% y`

`x %f!=% y`

`x %f<% y`

`x %f>% y`

`x %f<=% y`

`x %f>=% y`

```
x %f{}% bnd
```

```
x %f!{}% bnd
```

Arguments

x, y	numeric vectors, matrices, or arrays, though these operators were specifically designed for floats (class "double").
bnd	either a vector of length 2, or a matrix with 2 columns and 1 row, or else a matrix with 2 columns where <code>nrow(bnd)==length(x)</code> . The first element/column of bnd gives the lower bound of the closed interval; The second element/column of bnd gives the upper bound of the closed interval;

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

x <- c(0.3, 0.6, 0.7)
bnd <- matrix(c(0.29, 0.59, 0.69, 0.31, 0.61, 0.71), ncol=2)
x %f{}% bnd
x %f!{}% bnd

# 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

```

force_libPaths

Simple Project isolation by forcing library Paths

Description

The `force_libPaths()` function allows the user to force R to specific libraries. This may in some occasions be needed as base R's `.libPaths()` function only allows adding new library paths, not overwrite existing site or system libraries. The library paths are of course re-set again every time R restarts.

Usage

```
force_libPaths(lib_vec)
```

Arguments

`lib_vec` a character vector giving the new library path(s). Just like in `.libPaths()`, the order matters: R will first look for packages in the first path in `.libPaths()`, and if it cannot find the package(s), it will look for the packages in the second path in `.libPaths()`, etc.

Value

`force_libPaths()`: adjusts the R library paths as defined in `.libPaths()` directly.

References

McBain (2019, June 20). Before I Sleep: Hacking R's library paths. Retrieved from <https://milesmbain.com/posts/hacking-r-library-paths/>

Examples

```

## Not run:
force_libPaths("/mylibrary")

## End(Not run)

```

Description

These functions and operator are focussed on making it easier to use packages without having to explicitly attaching them to your namespace.

`import_as:`

The `import_as()` function imports the namespaces of an R package (or a small set of R packages that "belong" to each other) under the same alias.

`import_inops:`

The `import_inops()` function exposes the infix operators of the specified packages to the current environment (usually the global environment).

To ensure the user can still verify which operator function came from which package, a "package" attribute is added to each exposed operator.

Naturally, the namespaces of the operators remain intact.

`import_data:`

The `import_data()` function gets a specified data set from a package.

Unlike `utils::data()`, the `import_data()` function returns the data set directly, and allows assigning the data set like so:

```
mydata <- import_data(...).
```

`import_lsf:`

The `import_lsf(package, ...)` function gets a list of exported functions/operators from a package.

`pkgs %installed in% lib.loc:`

The `pkgs %installed in% lib.loc` operator checks if one or more package(s) `pkgs` exist(s) in library location `lib.loc`.

Now you no longer have to attach a package with `require()` simply to check if it exists.

Moreover, this operator makes it syntactically explicit in your code where you are looking for your R package(s).

Usage

```
import_as(alias, pkgs, lib.loc = .libPaths())
```

```
import_inops(pkgs, lib.loc = .libPaths(), exclude, include.only)
```

```
import_data(dataname, package, lib.loc = .libPaths())
```

```
import_lsf(package, type, lib.loc = .libPaths())
```

```
pkgs %installed in% lib.loc
```

Arguments

alias	a variable name (unquoted), giving the (not yet existing) object where the package(s) are to be assigned to. Syntactically invalid names are not allowed for the alias name.
pkgs	a single string, or character vector, with the package name(s). NOTE (1): The order of the character vector matters! If 2 packages share objects with the same name, the package named last will overwrite the earlier named package. NOTE (2): When supplying more than one package to <code>import_as()</code> , it is strongly advised to only import packages together under the same alias that are (reverse) dependencies of each other (i.e. they appear in each others Depends or Imports sections in the Description files). NOTE (3): Related to NOTE (2), the <code>import_as()</code> function only performs a very basic check for dependencies; the user is expected to use the <code>import_as()</code> function responsibly.
lib.loc	character vector specifying library search path (the location of R library trees to search through). This is usually <code>.libPaths()</code> . See also loadNamespace .
exclude	a character vector, giving the infix operators NOT to expose to the current environment. This can be handy to prevent overwriting any (user defined) infix operators already present in the current environment.
include.only	a character vector, giving the infix operators to expose to the current environment, and the rest of the operators will not be exposed. This can be handy to prevent overwriting any (user defined) infix operators already present in the current environment.
dataname	a single string, giving the name of the data set.
package	a single string, giving the name of the package.
type	The type of functions to list. Possibilities: "inops" or "operators": Only infix operators (functions surrounded by percentage signs). "regfuns": Only regular functions (thus excluding infix operators). "all": All functions, both regular and infix operators.

Details

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

The `import_as` and `import_inops` functions will inform the user about conflicting objects.

The `import_as` function will give a warning when more than 3 packages being imported into the same alias.

Value

For `import_as`:

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

If argument `expose_inops = TRUE`, the infix operators of the specified package(s) are placed in the current environment.

For `import_data()`:

Returns the data directly. Thus, one can assign the data like so: `mydata <- import_data(...)`.

For `import_lsf()`:

Returns a character vector of function and/or operator names.

For `pkgs %installed in% lib.loc`:

Returns a logical vector, where `TRUE` indicates a package is installed, and `FALSE` indicates a package is not installed.

Examples

```
## Not run:
pkgs <- c(unlist(tools::package_dependencies("devtools")), "devtools")
pkgs %installed in% .libPaths()
import_as(devt, pkgs) # this creates the devt object
import_inops(pkgs)
d <- import_data("chicago", "gamair")
head(d)

## End(Not run)
```

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 %alog =% 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 %alog =% b`

Arguments

<code>x</code>	a number or numeric (or 'number-like') vector, matrix, or array.
<code>y</code>	a number, or numeric (or 'number-like') vector, matrix, or array of the same length/dimension as <code>x</code> . It gives the number to add, subtract, multiply by, or divide by.
<code>p</code>	a number, or a numeric vector of the same length as <code>x</code> . It gives the power to be used.
<code>b</code>	a number, or a numeric vector of the same length as <code>x</code> . 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 %alog 3 # same as x <- 3^x
print(x)

x <- 3
print(x)
x %alog 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](#).

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	<i>In place modifying string subsetting</i>
--------------------	---

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)
```

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%&% y` 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 be 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

<code>x, y</code>	see Logic .
<code>s</code>	a character vector.
<code>p</code>	the result from s_pattern , or else a character vector of the same length as <code>s</code> with regular expressions.
<code>n</code>	a numeric vector.
<code>numtype</code>	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 actual number 5.0.
- "special": checks if the string consists of only special characters.

Examples

```
x <- c(TRUE, FALSE, TRUE, FALSE, NA, NaN, Inf, -Inf, TRUE, FALSE)
y <- c(FALSE, TRUE, TRUE, FALSE, rep(NA, 6))
outcome <- data.frame(
  x=x, y=y,
  "x %xor% y"=x %xor% y, "x %n&% y" = x %n&% y, "x %?=% y" = x %?=% y,
  check.names = FALSE
)
print(outcome)
```

```
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

*Infix operators for row- and column-wise re-ordering of matrices***Description**

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

The `x %row~% mat` operator re-orders the elements of every row of matrix `x` according to the ordering ranks given in matrix `mat`.

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

Usage

```
x %row~% mat
```

```
x %col~% mat
```

Arguments

`x` a matrix

`mat` a matrix with the same dimensions as `x`, giving the ordering ranks 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, sorting the elements using `x %row~% x` and `x %col~% x` are still possible, but probably not the best option. In the non-numeric case, providing a matrix of ordering ranks for `mat` would be faster and give more accurate ordering. See the examples section.

If `mat` is a matrix of non-repeating random integers (i.e. `sample(1:length(x), replace=FALSE)` `sample(1:length(x)) |> matrix(ncol=ncol(x))`), `x %row~% mat` will randomly shuffle the elements of every row, where the shuffling order of every row is independent of the other rows.

Similarly, `x %col~% mat` will randomly shuffle the elements of every column, where the shuffling order of every column is independent of the other columns.

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 ====

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

x <- matrix(sample(1:25), nrow=5)
print(x)
mat <- sample(1:length(x)) |> matrix(ncol=ncol(x)) # matrix of non-repeating random integers
x %row~% mat # randomly shuffle every row independently
x %col~% mat # randomize shuffle every column independently

# character matrix ====

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

x <- matrix(sample(letters, 25), nrow=5)
print(x)
mat <- sample(1:length(x)) |> matrix(ncol=ncol(x)) # matrix of non-repeating random integers
x %row~% mat # randomly shuffle every row independently
x %col~% mat # 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 <- stringi::stri_rank(as.vector(mat)) |> matrix(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 <- stringi::stri_rank(as.vector(mat)) |> matrix(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	<i>Locate i^{th} Pattern Occurrence</i>
-----------------	--

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)
```

Arguments

<code>str</code>	a string or character vector.
<code>i</code>	<p>a number, or a numeric vector of the same length as <code>str</code>. This gives the i^{th} instance to be replaced.</p> <p>Positive numbers are counting from the left. Negative numbers are counting from the right. I.e.:</p> <p><code>stri_locate_ith(str, i=1, ...)</code> gives the position (range) of the first occurrence of a pattern.</p> <p><code>stri_locate_ith(str, i=-1, ...)</code> gives the position (range) of the last occurrence of a pattern.</p> <p><code>stri_locate_ith(str, i=2, ...)</code> gives the position (range) of the second occurrence of a pattern.</p> <p><code>stri_locate_ith(str, i=-2, ...)</code> gives the position (range) of the second-last occurrence of a pattern.</p> <p>If <code>abs(i)</code> is larger than the number of instances, the first (if <code>i < 0</code>) or last (if <code>i > 0</code>) instance will be given.</p> <p>For example: suppose a string has 3 instances of some pattern;</p> <p>then if <code>i >= 4</code> the third instance will be located,</p> <p>and if <code>i <= -3</code> the first instance will be located.</p>
<code>...</code>	more arguments to be supplied to stri_locate and stri_count .
<code>regex, fixed, coll, charclass</code>	a character vector of search patterns, as in stri_locate .

Value

The `stri_locate_ith()` function returns an integer matrix with two columns, giving the start and end positions of the i^{th} matches, and two NAs if no matches are found.

Examples

```
# simple pattern ====

x <- rep(paste0(1:10, collapse=""), 10)
print(x)
out <- stri_locate_ith(x, 1:10, regex = as.character(1:10))
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)
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)
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, 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")
extr <- stringi::stri_sub(x, from=loc)
repl <- chartr(extr, old = "a-zA-Z", new = "A-Za-z")
stringi::stri_sub_replace(x, loc, replacement=repl)
```

str_arithmetic

*String arithmetic***Description**

String arithmetic operators.

The `x %s+% y` operator is equivalent to `stringi::stri_c(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

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

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 <code>nrow(ss)==length(x)</code> . The object <code>ss</code> 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 <code>ss</code> gives the number of characters counting from the left side to be extracted/removed from <code>x</code> . The second element/column of <code>ss</code> gives the number of characters counting from the right side to be extracted/removed from <code>x</code> .

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

*Substr - 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 reverses 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. 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, giving the position after or before which the string is to be added.
type	a single string, giving the part of the string to extract. 3 options available: <ul style="list-style-type: none"> • 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("goodGOODGoodgO0d", "goodGOODGoodgO0d", 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(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`, `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), case_insensitive=TRUE)
x %s/% p # count how often vowels appear in each string of vector x.
```

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 additional logic operators, safer float (in)equality operators, and infix operators for custom row- and column-wise ordering of matrices. It also adds some `stringi`-based string related functions and operators. It also adds operators and a few functions to help reduce unnecessary repetitive code. And finally, it also adds some functions and an operator for easier package/library management. The 'tidyoperators' R-package has only one dependency, namely 'stringi', though it does allow multi-threading of some of the string-related functions (when appropriate) via the suggested 'stringfish' R-package.

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](#).
- [Infix operators for row- and column-wise rank-based 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.
- [New package import functions](#).
- 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 before using this package:

<https://github.com/tony-aw/tidyoperators>

Usage

```
tidyoperators_help()
```

transform_if	<i>The transform_if function and the subset_if operators</i>
--------------	--

Description

Consider the following code:

```
ifelse(cond(x), f(x), g(x))
```

Here a conditional subset of the object `x` is transformed, where the condition is using a function referring to `x` itself. Consequently, reference to `x` is written four times!

The `tidyoperators` package therefore adds the `transform_if()` function which will tidy this up.

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.

The `tidyoperators` package also adds the `x %unreal =% repl` operator:

`x %unreal =% repl` is the same as `x[is.na(x)|is.nan(x)|is.infinite(x)] <- repl`

Usage

```
transform_if(x, cond, trans_T = function(x) x, trans_F = function(x) x)
```

```
x %[if]% cond
```

```
x %[!if]% cond
```

```
x %unreal =% repl
```

Arguments

<code>x</code>	a vector, matrix, or array.
<code>cond</code>	a function that returns a binary logic (TRUE, FALSE) vector of the same length/dimensions as <code>x</code> (for example: <code>is.na</code>).
<code>trans_T</code>	the transformation function to use when <code>cond(x)==TRUE</code> . For example: <code>log</code> . If this is not specified, <code>trans_T</code> defaults to <code>function(x)x</code> .
<code>trans_F</code>	the transformation function to use when <code>cond(x)==FALSE</code> . For example: <code>log</code> . If this is not specified, <code>trans_F</code> defaults to <code>function(x)x</code> .
<code>repl</code>	the replacement value.

Details

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

Value

For `transform_if()`:

Similar to [ifelse](#). However, unlike `ifelse()`, the transformations are evaluated as `trans_T(x[cond(x)])` and `trans_F(x[!cond(x)])`, ensuring no unnecessary warnings or errors occur.

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

The `x %unreal =% repl` operator does not return any value:

It is an in-place modifiers, and thus modifies `x` directly. The object `x` is modified such that all NA, NaN and Inf elements are replaced with `repl`.

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 |> transform_if(\(x)x>0, log, \(x)x^2)
object_with_very_long_name %[if]% \(x)x %in% 1:10
object_with_very_long_name %[!if]% \(x)x %in% 1:10

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)
```

Index

*** join_mat**
 stri_join_mat, 16
%* =(inplace_math), 7
%+ =(inplace_math), 7
%- =(inplace_math), 7
%/ =(inplace_math), 7
%=numtype%(logic_ops), 12
%=strtype%(logic_ops), 12
%?=(logic_ops), 12
%[!if]%(transform_if), 29
 %[if]%(transform_if), 29
%^ =(inplace_math), 7
%alog =(inplace_math), 7
%col~%(matrix_ops), 15
%f!=%(float_logic), 2
%f<=(float_logic), 2
%f<%(float_logic), 2
%f==%(float_logic), 2
%f>=(float_logic), 2
%f>%(float_logic), 2
%installed in%(import), 5
%logb =(inplace_math), 7
%n&%(logic_ops), 12
%out%(logic_ops), 12
%row~%(matrix_ops), 15
%rt =(inplace_math), 7
%s* =(inplace_str_arithmetic), 9
%s*%(str_arithmetic), 20
%s+ =(inplace_str_arithmetic), 9
%s+%(str_arithmetic), 20
%s- =(inplace_str_arithmetic), 9
%s-%(str_arithmetic), 20
%s/ =(inplace_str_arithmetic), 9
%s/%(str_arithmetic), 20
%sget =(inplace_str_subset), 11
%sget%(str_subset_ops), 21
%sgrep%(logic_ops), 12
%ss%(str_subset_ops), 21
%strim =(inplace_str_subset), 11
%strim%(str_subset_ops), 21
%unreal =(transform_if), 29
%xor%(logic_ops), 12

chartr, 24

float_logic, 2
force_libPaths, 4

ifelse, 30
import, 5
import_as(import), 5
import_data(import), 5
import_inops(import), 5
import_lsf(import), 5
In-place modifiers for mathematical
 arithmetic, 28
In-place modifying string arithmetic,
 28
In-place modifying string sub-setting,
 28
Infix logical operators, 28
Infix operators for row- and
 column-wise rank-based
 re-ordering of matrices, 28
inplace_math, 7
inplace_str_arithmetic, 9
inplace_str_subset, 11

loadNamespace, 6
Logic, 13
logic_ops, 12

matrix_ops, 15

New package import functions, 28

s_pattern, 10, 13, 20, 26, 28
Safer (in)equality operators for
 floating numbers, 28
str_arithmetic, 20
str_subset_ops, 21
stri_c_mat(stri_join_mat), 16
stri_count, 18
stri_join, 17
stri_join_mat, 16, 28
stri_locate, 18
stri_locate_ith, 18, 24, 28
stri_paste_mat(stri_join_mat), 16
stri_rank, 25
string arithmetic, 10, 28

string sub-setting, [28](#)
string subset, [11](#)
sub-string functions, [28](#)
substr_addin(substr_repl), [23](#)
substr_arrange(substr_repl), [23](#)
substr_chartr(substr_repl), [23](#)
substr_extract(substr_repl), [23](#)
substr_repl, [23](#)

tidyoperators_help, [27](#)
transform_if, [28](#), [29](#)