Package 'tinyoperators'

June 20, 2023

Title Infix operators and some functions to help in proper coding etiquette

Version 0.0.9

Description The 'tinyoperators' R-package adds some much needed infix operators,
and a few functions, to make your R code much more tiny.
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 'tinyoperators' R-package has only one dependency, namely 'stringi',
though it does allows 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/tinyoperators

BugReports https://github.com/tony-aw/tinyoperators/issues

R topics documented:

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

Description

The `%f==%`, `%f!=%` `%f<%`, `%f>%`, `%f<=%`, `%f>=%` (in)equality operator perform float truth testing. They are virtually equivalent to the regular (in)equality operators,
`==`, `!=`, `<`, `>`, `<=`, `>=`,
except for one aspect. The float truth testing 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 truth testing.
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 `nrow(bnd)==length(x)`. 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
```

`import` *Additional package import management*

Description

These functions and operator are focused 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 (like the global environment, or the environment within a function).
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 multiple 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 `import_as()`, 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 `import_as()` function only performs a very basic check for dependencies; the user is expected to use the `import_as()` function responsibly.
lib.loc	character vector specifying library search path (the location of R library trees to search through). This is usually `.libPaths()`. 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. `"regfuns"`: Only regular functions (thus excluding infix operators). `"all"`: All functions, both regular functions 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.

For `import_inops()`:
The infix operators from the specified packages will be placed in the current environment (like the Global environment, or the environment within a function).

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 named 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 *Generalized in-place (mathematical) modifier*

Description

Generalized in-place (mathematical) modifier.

The `x %:=% f` operator allows performing in-place modification of some object `x` with a function `f`.

For example this:

`object[object > 0] %:=% \(x) x + 1`
Is the same as:
`object[object > 0] <- object[object > 0] + 1`

This function-based method is used instead of the more traditional in-place mathematical modification like `+=` to prevent precedence issues (functions come before mathematical arithmetic in R).

Usage

```
x %:=% f
```

Arguments

`x`	an object, with properties such that function `f` can be use on it. For example, when function `f` is mathematical, `x` should be a number or numeric (or 'number-like') vector, matrix, or array.
`f`	a function to be applied in-place on `x`.

Value

This operator does not return any value:
it is an in-place modifiers, and thus modifies `x` directly.

Examples

```
set.seed(1)
object <- matrix(rpois(10, 10), ncol=2)
print(object)
object %:=% \(x) x+3 # same as object <- object + 3
print(object)
```

---

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` *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)
```

`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 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, 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
```

---

source_module *Additional module import management*

Description

The `alias %source from% list(file=...)` operator imports all objects from a source-able script file under an `alias`.

The `source_inops()` function exposes the infix operators defined in a source-able script file to the current environment (like the global environment, or the environment within a function).

Note that the `alias %source from% list(file=...)` operator and the `source_inops()` function do NOT suppress output (i.e. plots, prints, messages) from the sourced module file.

Usage

```
alias %source from% lst

source_inops(...)
```

Arguments

alias	a variable name (unquoted), giving the (not yet existing) object where the sourced objects from the module are to be assigned to. Syntactically invalid names are not allowed for the alias name.
lst	a named list, giving the arguments to be passed to the source function. For example: `alias %source from% list(file="mydir/myscript.R")`
...	arguments to be passed to the source function, such as the `file` argument.

Value

For the `alias %source from% list(file=...)` operator:
The variable named as the `alias` will be created (if it did not already exist) in the current environment, and will contain all objects from the sourced script.

For `source_inops()`:
The infix operators from the specified module will be placed in the current environment (like the Global environment, or the environment within a function).

Examples

```
## Not run:
alias %source from% list(file="mydir/mymodule.R")
source_inops(file="mydir/mymodule.R")

## End(Not run)
```

---

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` *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)
```

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, ...)` gives the position (range) of the first occurrence of a pattern. `stri_locate_ith(str, i=-1, ...)` gives the position (range) of the last occurrence of a pattern. `stri_locate_ith(str, i=2, ...)` gives the position (range) of the second occurrence of a pattern. `stri_locate_ith(str, i=-2, ...)` gives the position (range) of the second-last occurrence of a pattern. 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 some pattern; then if `i >= 4` the third instance will be located, and if `i <= -3` the first instance will be located.
`...`	more arguments to be supplied to stri_locate and stri_count.
`regex, fixed, coll, charclass`	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 `NA`s 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

`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 *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 tinyoperators 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: • 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(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, 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.
```

`tinyoperators_help` *The tinyoperators help page*

Description

Welcome to the `tinyoperators` help page!

The 'tinyoperators' R-package adds some much needed infix operators, and a few functions, to make your R code much more tiny. 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 'tinyoperators' R-package has only one dependency, namely 'stringi', though it does allows multi-threading of some of the string-related functions (when appropriate) via the suggested 'stringfish' R-package.

The `tinyoperators` 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 string arithmetic.
- Infix operators for string sub-setting.
- Several operators for the "Don't Repeat Yourself" coding principle (`DRY`). This includes the generalized in-place (mathematical) modification operator, infix operators for In-place modifying string arithmetic, and infix operators for In-place modifying string sub-setting.
- Infix operators for row- and column-wise re-ordering of matrices.
- The `tinyoperators` 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 i^{th} pattern occurrence or location.
- The s_pattern helper function for string operators.
- New package import functions, and new module sourcing 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 the sub-string functions through the `stringfish` R package.

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

Attaching a package necessarily loads a package, but loading a package does not entail attaching a package.

Whenever a package is used in any way, even internally by a function, a package is **loaded**. A loaded package is simply a package that R is prepared to use. A package that is loaded is not necessarily attached. But **attaching** package is what happens when one uses the `library()` or `require()` function:
the functions from the package are exposed to the namespace.
Exposing functions to the namespace is **not a necessity**, it is merely a **convenience**.

Without attaching a package, there are basically 2 ways to use functions from a package:

- Using the `::` operator, like so: `package_name::function_name()`.

- If a package name is very long, the `::` approach may eventually become annoying. So alternatively, one can use an abbreviated `alias`, like so: `alias <- loadNamespace("packagename")`; then one can use `alias$function_name()`, instead of the longer `package_name::function_name()`.

The advantages of attaching a package instead are as follows:

1. **Less typing**: You have to type less to use functions. This is especially handy for **infix operators** - which `tinyoperators` obviously focuses on - as operators use special characters, which require them to be surrounded by back-ticks when using `::` or `alias$`.

2. **More collective usage**: If multiple packages are meant to work together, constantly switching between the different package name/alias prefixes may eventually become annoying and even syntactically chaotic-looking. By attaching the packages, you no longer have to deal with this annoyance.

But, attaching many packages willy-nilly has a few potentially serious drawbacks:

1. **masking namespaces**: functions from different packages that have the same function names will mask each other.
2. **overloading core R functions**: some R packages overload core R functions. This is not always wanted (or even expected) from the user. Using `::` or a package alias will allow one to explicitly choose whether to use the original R function, or the package function.
3. **lack of code clarity**: The absence of a package name or alias prefix at function calls makes it less clear from which package which function came. This will be compounded when an R package overloads core R functions. If something goes wrong in a script, one has to figure out from which package a function came from, in order to figure out the cause of the issue. The lack of a package name or alias prefix makes this more cumbersome.
4. **global assignment**: Unlike aliases, which exist **locally**, when packages are attached, they are attached **globally**. Using `library()` inside a function will thus not merely attach the package inside the function, but attach it globally, which may worsen the previously named problems when a function silently attaches a package.
5. **Polluting your namespace**: The more packages one attaches, the greater the chance for bugs caused by masked namespaces, and the greater the difficulty in de-bugging your code due to the aforementioned lack of syntactical clarity.

Some programming languages don't even allow attaching packages, because of these issues.

I do see the advantages of attaching a package - especially when it comes to using `infix operators`. But the disadvantages of attaching a package cannot just be ignored. So what `tinyoperators` attempts to do with the functions described in this section, is to somewhat find the best of both worlds. Basically, `tinyoperators` has functions that allow the following functionality lacking in base R:

- Allow **multiple related** packages to be loaded under **one alias**. This essentially combines the advantages of "collective usage" (see attaching advantage number 2), whilst keeping most advantages of only loading a package under an alias.
- Allow **exposing** of infix operators to the **current environment**. This gains advantages "less typing" whilst simultaneously avoiding the disadvantage of "global assignment".

Moreover, `tinyoperators` extends this functionality to also work on **sourced modules**.

Usage

```
tinyoperators_help()

tinyoperators_importsystem()
```

`transform_if` *The transform_if function and the subset_if operators*

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 `tinyoperators` package therefore adds the `transform_if()` function which will tiny this up.

The `tinyoperators` 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 `tinyoperators` 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

`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`).
`trans_T`	the transformation function to use when `cond(x)==TRUE`. For example: `log`. If this is not specified, `trans_T` defaults to `function(x)x`.
`trans_F`	the transformation function to use when `cond(x)==FALSE`. For example: `log`. If this is not specified, `trans_F` defaults to `function(x)x`.
`repl`	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