# Package 'tinyoperations'

July 22, 2023

**Title** Functions and infix operators to help in your programming etiquette

**Version** 0.0.0.9

**Description** The 'tinyoperations' R-package adds some infix operators, and a few functions.
It primarily focuses on 4 things.
(1) Safer decimal numbers (``double") truth testing.
(2) Extending the string manipulation capabilities of the 'stringi' R package.
(3) Reducing repetitive code.
(4) A new package and module import system, that combines the benefits of aliasing a package with the benefits of attaching a package.
The 'tinyoperations' R-package has only one dependency, namely 'stringi'.
Most functions in this R-
package are fully vectorized and have been optimized for optimal speed and performance.

**License** MIT + file LICENSE

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**RoxygenNote** 7.2.3

**Suggests** rlang,
knitr,
rmarkdown,
tinytest,
pkgdown,
devtools,
fastverse

**VignetteBuilder** knitr

**Depends** R (>= 4.1.0)

**Imports** stringi (>= 1.7.12)

**URL** https://github.com/tony-aw/tinyoperations

**BugReports** https://github.com/tony-aw/tinyoperations/issues

## R topics documented:

---

`atomic_conversions` *Safer atomic type casting*

---

## Description

Atomic type casting in R is generally performed using the functions as.logical, as.integer, as.double, as.character.
There are a few annoying aspect of R with respect to atomic type casting in R using these functions:

- converting an object between atomic types strips the object of its attributes, including attributes such as names and dimensions.
- the conversions are somewhat inconsistent. For example, to prevent stripping attributes, one can do something like this:
  `x[] <- as.numeric()`
  but this is not always the same as first converting the object and then re-assigning the attributes.

The functions provided here by the `tinyoperations` package do not strip strip away attributes.

The functions are as follows:

- `as_bool()`: converts object to class `logical` (`TRUE`, `FALSE`).
- `as_int()`: converts object to class `integer`.
- `as_dbl()`: converts object to class `double` (AKA decimal numbers).
- `as_chr()`: converts object to class `character`.

Moreover, the function `is_wholenumber()` is added, to safely test for whole numbers.

## Usage

```
as_bool(x, ...)

as_int(x, ...)

as_dbl(x, ...)

as_chr(x, ...)

is_wholenumber(x, tol = .Machine$double.eps^0.5)
```

## Arguments

| | |
|---|---|
| x | vector, matrix, array (or similar object where all elements share the same `atomic class`), to be converted to some other `atomic class`. |
| ... | further arguments passed to or from other methods. |
| tol | the tolerance. |

## Value

The converted object.

## See Also

tinyoperations_dry

## Examples

```
x <- rnorm(10) |> matrix(ncol=2)
colnames(x) <- c("one", "two")
attr(x, "test") <- "test"

# notice that in all following, attributes are conserved:
as_bool(x)
as_int(x)
as_dbl(x)
as_chr(x)
```

---

`decimal_truth` *Safer decimal number (in)equality testing operators*

---

## Description

The `%d==%`, `%d!=%` `%d<%`, `%d>%`, `%d<=%`, `%d>=%` (in)equality operator perform decimal (class "double") number truth testing.
They are virtually equivalent to the regular (in)equality operators,
`==`, `!=`, `<`, `>`, `<=`, `>=`,
except for one aspect:
The decimal number (in)equality operators assume that if the absolute difference between any two

numbers x and y is smaller than the Machine tolerance, sqrt(.Machine$double.eps), then x and y should be consider to be equal.

Thus these operators provide safer decimal number (in)equality tests.

For example: 0.1*7 == 0.7 returns FALSE, even though they are equal, due to the way decimal numbers are stored in programming languages (like R). But 0.1*7 %d==% 0.7 returns TRUE.

There are also the x %d{}% bnd and x %d!{}% 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 %d{}% bnd operator checks if x is within the closed interval with bounds defined by bnd.
The x %d!{}% bnd operator checks if x is outside the closed interval with bounds defined by bnd.

## Usage

```
x %d==% y

x %d!=% y

x %d<% y

x %d>% y

x %d<=% y

x %d>=% y

x %d{}% bnd

x %d!{}% bnd

tinyoperations_decimal_truth()
```

## Arguments

| | |
|---|---|
| x, y | numeric vectors, matrices, or arrays, though these operators were specifically designed for decimal numbers (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).<br>The first element/column of bnd gives the lower bound of the closed interval;<br>The second element/column of bnd gives the upper bound of the closed interval; |

## Value

Same as ==.

## See Also

tinyoperations_help

## 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 %d==% y # here it's done correctly
x %d!=% y # correct
x %d<% y # correct
x %d>% y # correct
x %d<=% y # correct
x %d>=% 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 %d{}% bnd
x %d!{}% bnd

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

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

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

---

`import_as` *Load main package + its foreign exports + its dependencies + its enhances + its extensions under one alias*

---

## Description

The `import_as()` function imports the namespace of an R package, and optionally also its dependencies, enhances, and extensions, under the same alias. The specified alias will be placed in the current environment (like the global environment, or the environment within a function).

## Usage

```
import_as(
  alias,
  main_package,
  foreign_exports = TRUE,
  dependencies = NULL,
  enhances = NULL,
  extensions = NULL,
  lib.loc = .libPaths(),
  verbose = FALSE,
  loadorder = c("dependencies", "main_package", "enhances", "extensions")
)
```

## Arguments

| | |
|---|---|
| `alias` | a syntactically valid non-hidden object name (unquoted), giving the alias object where the package(s) are to be loaded into.<br>NOTE: To keep aliases easily distinguishable from other objects that can also be subset with the `$` operator, I recommend ending (not starting!) the names of all alias names with a dot (`.`) or underscore (`_`). |
| `main_package` | a single string, giving the name of the main package to load under the given alias. |
| `foreign_exports` | logical; some R packages export functions that are not defined in their own package, but in their direct dependencies; "foreign exports", if you will.<br>This argument determines what the `import_as` function will do with the foreign exports of the `main_package`:<br>• If `TRUE` the foreign exports from the `main_package` are added to the alias, even if `dependencies = NULL`. This is the default, as it is analogous to the behaviour of base R's `::` operator.<br>• If `FALSE`, these foreign exports are not added, and the user must specify the appropriate packages in argument `dependencies`. |
| `dependencies` | an optional character vector, giving the names of the dependencies of the `main_package` to be loaded also under the alias.<br>Defaults to `NULL`, which means no dependencies are loaded.<br>See pkg_get_deps to quickly get dependencies from a package. |
| `enhances` | an optional character vector, giving the names of the packages enhanced by the `main_package` to be loaded also under the alias.<br>Defaults to `NULL`, which means no enhances are loaded. |
| `extensions` | an optional character vector, giving the names of the extensions of the `main_package` to be loaded also under the alias.<br>Defaults to `NULL`, which means no extensions are loaded. |

`lib.loc` character vector specifying library search path (the location of R library trees to search through).
This is usually `.libPaths()`.
See also loadNamespace.

`verbose` logical, indicating whether messages regarding conflicts and foreign exports should be printed while importing packages (`TRUE`), or if these should not be printed (`FALSE`).
Defaults to `FALSE`, because all information conveyed by the messages can be more compactly be viewed by viewing the attributes of the alias object (see attributes).

`loadorder` the character vector
`c("dependencies", "main_package", "enhances", "extensions")`,
or some re-ordering of this character vector, giving the relative load order of the groups of packages.
The default setting (which is highly recommended) is the character vector
`c("dependencies", "main_package", "enhances", "extensions")`,
which results in the following load order:

1. The dependencies, **in the order specified by the** `depenencies` **argument**.
2. The main_package (see argument `main_package`), including foreign exports (if `foreign_exports=TRUE`).
3. The enhances, **in the order specified by the** `enhances` **argument**.
4. The extensions, **in the order specified by the** `extensions` **argument**.

## Details

**On the** `dependencies`**,** `enhances` **and** `extensions` **arguments**

- `dependencies`: "Dependencies" here are defined as any package appearing in the "Depends", "Imports", or "LinkingTo" fields of the Description file of the `main_package`. So no recursive dependencies.
- `enhances`: Enhances are defined as packages appearing in the "Enhances" field of the Description file of the `main_package`.
- `extensions`: "Extensions" here are defined as reverse-depends or reverse-imports. It does not matter if these are CRAN or non-CRAN packages. However, the intended meaning of an extension is not merely being a reverse dependency, but a package that actually extends the functionality of the `main_package`.

As implied in the description of the `loadorder` argument, the order of the character vectors given in the `dependencies`, `enhances`, and `extensions` arguments matter:
If multiple packages share objects with the same name, the objects of the package named last will overwrite those of the earlier named packages.

### Additional details

The `import_as()` function does not allow importing base/core R under an alias, so don't try.

## Value

A locked package alias object as named in the `alias` argument will be created in the current environment (like the global environment, or the environment within a function). The alias object will contain the following:

- The (merged) package environment, containing the exported functions from the packages.
- The attributes of the alias object will contain the package order, input arguments, and a conflicts report.

To use, for example, function "some_function()" from alias "alias.", use:
`alias.$some_function()`

For information on the binding lock used, see import_lock.

## See Also

`tinyoperations_import()`

## Examples

```
## Not run:
import_as( # this creates the 'tdt.' object
  tdt., "tidytable", dependencies = "data.table"
)
tdt.$mutate

## End(Not run)
```

---

# import_data *Assign data-set from a package directly to a variable*

---

## Description

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(...)`.

## Usage

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

## Arguments

| | |
|---|---|
| `dataname` | a single string, giving the name of the data set. |
| `package` | the quoted package name. |
| `lib.loc` | character vector specifying library search path (the location of R library trees to search through).<br>This is usually `.libPaths()`.<br>See also loadNamespace. |

## Value

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

## See Also

`tinyoperations_import()`

## Examples

```
## Not run:
d <- import_data("chicago", "gamair")
head(d)

## End(Not run)
```

---

`import_inops` *Expose infix operators to the current environment*

---

## Description

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.
If you wish to globally attach infix operators, instead of just placing them in the current environment, see pkg_lsf.

## Usage

```
import_inops(
  pkgs,
  lib.loc = .libPaths(),
  exclude,
  include.only,
  overwrite = TRUE,
  inherits = FALSE,
  delete
)
```

## Arguments

| | |
|---|---|
| `pkgs` | a character vector of package name(s) from which to load and expose infix operators.<br>NOTE: The order of the character vector matters! If multiple packages share infix operators with the same name, the conflicting operators of the package named last will overwrite those of the earlier named packages. |

`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.

`overwrite` logical, indicating if it is allowed to overwrite existing infix operators.

- If `TRUE` (default), a warning is given when operators existing in the current environment are being overwritten, but the function continuous nonetheless.
- If `FALSE`, an error is produced when the to be exposed operators already exist in the current environment, and the function is halted.

`inherits` logical; when `overwrite=FALSE`, should enclosed environments, especially package namespaces, also be taken into account?
Defaults to `FALSE`.
See also exists.

`delete` a character vector of package name(s) from which to remove infix operators.
Normally, the `import_inops()` function exposes infix operators from packages `pkgs` to the current environment.
But if the user specifies package names in the `delete` argument, the `import_inops()` function will ignore all other arguments, and simply delete all infix operators from the named packages exposed by the `import_inops()` function.
Infix operators manually defined by the user themselves will not be touched.

## Details

**On the `exclude` and `include.only` arguments**:
You cannot specify both the `exclude` and `include.only` arguments. Only one or the other, or neither.

**On the `pkgs` and `delete` arguments**:
You cannot specify both the `pkgs` and `delete` arguments. Only one or the other.
Unlike the import_as function, the `import_inops()` function does not require the packages to be necessarily related to each other.

**Other details**:
The `import_inops()` function does not support overloading base/core R operators, so don't try.

### Value

If pkgs is specified:
The infix operators from the specified packages will be placed in the current environment (like the Global environment, or the environment within a function).
The infix operators will be LOCKED. For information on the binding lock used, see import_lock.

If delete is specified:
The infix operators from the packages specified in delete, exposed by import_inops(), will be deleted.
If such infix operators could not be found, this function returns NULL.

### See Also

tinyoperations_import()

### Examples

```
## Not run:
import_inops("data.table") # expose infix operators
import_inops(delete="data.table") # remove the exposed infix operators.

## End(Not run)
```

---

## import_lock *Explanation of tinyoperations' Import Lock*

---

### Description

The import_as function returns a locked environment, and the import_inops function returns one or more locked infix operators.
These returned objects are locked to prevent the user from (accidentally) messing with the functions, or their attributes.
The import_inops() and import_as() functions will delete locked objects created by import_inops() or import_as() when it needs to overwrite them. Any user-defined objects are not touched in this process. The attributes of the objects, as well as class, namespace names, and other properties, are used to determine if any object is created by import_as / import_inops or not. This should be quite safe.

### Usage

```
import_lock()
```

### See Also

tinyoperations_import()

`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:
`mtcars$mpg\[mtcars$cyl>6\] <- mtcars$mpg\[mtcars$cyl>6\]^2`
Can now be re-written as:
`mtcars$mpg[mtcars$cyl>6] %:=% \(x)x^2`

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.<br>For example, when function `f` is mathematical, `x` should be a number or numeric (or 'number-like') vector, matrix, or array. |
| `f` | a (possibly anonymous) function to be applied in-place on `x`. The function must take one argument only. |

## Value

This operator does not return any value:
It is an in-place modifiers, and thus modifies the object directly.

## See Also

`tinyoperations_dry()`

## Examples

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

Some of the internal code of these operators was inspired by the `roperators` R package.

## 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.

## References

Wiseman B, Nydick S, Jones J (2022). roperators: Additional Operators to Help you Write Cleaner R Code. https://CRAN.R-project.org/package=roperators

## See Also

`tinyoperations_dry()` `tinyoperations_stringi()`

## 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%`.

Some of the internal code of these operators was inspired by the `roperators` R package.

## 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.

## References

Wiseman B, Nydick S, Jones J (2022). roperators: Additional Operators to Help you Write Cleaner R Code. https://CRAN.R-project.org/package=roperators

## See Also

`tinyoperations_dry()` `tinyoperations_stringi()`

## 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&% ` 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 numeric type to be checked. The following options are supported:<br>• `"~0"`: zero, or else a number whose absolute value is smaller than the Machine tolerance (`sqrt(.Machine$double.eps)`).<br>• `"B"`: binary numbers (exactly 0 or exactly 1);<br>• `"prop"`: proportions - numbers between 0 and 1 (exactly 0 or 1 is also allowed);<br>• `"I"`: Integers;<br>• `"odd"`: odd integers;<br>• `"even"`: even integers;<br>• `"R"`: Real numbers;<br>• `"unreal"`: infinity, NA, or NaN; |
| `strtype` | a single string giving the string type to be checked. The following options are supported:<br>• `"empty"`: checks if the string only consists of empty spaces.<br>• `"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.<br>• `"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`.<br>• `"special"`: checks if the string consists of only special characters. |

## Value

A logical vector.

## 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% "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, each row ordered independently from the other rows, of matrix `x`, according to the ordering ranks given in matrix `mat`.

The `x %col~% mat` operator re-orders the elements of every column, each column ordered independently from the other columns, 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 is 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.
mat <- sample(1:length(x)) |> matrix(ncol=ncol(x)))
then the code
x %row~% mat
will randomly shuffle the elements of every row, where the shuffling order in each row is independent from the shuffling order in the other rows.
Similarly,
x %col~% mat
will randomly shuffle the elements of every column, where the shuffling order in each column is independent from the shuffling order in the other columns.

Re-ordering/sorting every row/column of a matrix with these operators is generally faster than doing so through loops or apply-like functions.

## Value

A modified matrix.

## See Also

tinyoperations_misc()

## Examples

```
# numeric matrix ====

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

```
x %col~% -x # reverse-sort elements of every column independently

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 independently
x %row~% -mat # reverse-sort elements of every row independently
x %col~% mat # sort elements of every column independently
x %col~% -mat # reverse-sort elements of every column independently

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
```

---

`misc` *Miscellaneous functions to help your coding etiquette*

---

## Description

The `lock_TF()` function locks the `T` and `F` values and sets them to `TRUE` and `FALSE`, to prevent the user from re-assigning them.
Removing the created `T` and `F` objects allows re-assignment again.

The `X %<-c% A` operator creates a `constant` `X` and assigns `A` to it.
Constants cannot be changed, only accessed or removed. So if you have a piece of code that requires some unchangeable `constant`, use this operator to create said `constant`.
Removing constant `X` also removes its binding lock. Thus to change a `constant`, simply remove it and re-create it.

## Usage

```
lock_TF()

X %<-c% A
```

## Arguments

| | |
|---|---|
| `X` | a syntactically valid unquoted name of the object to be created. |
| `A` | any kind of object to be assigned to `X`. |

## Value

For `lock_TF()`:
Two `constants`, namely `T` and `F`, set to `TRUE` and `FALSE` respectively, are created in the current environment, and locked. Removing the created `T` and `F` objects allows re-assignment again.

For `X %<-c% A`:
The object `X` containing `A` is created in the current environment, and this object cannot be changed. It can only be accessed or removed.

## See Also

`tinyoperations_misc()`

## Examples

```
lock_TF()
X %<-c% data.frame(x=3, y=2) # this data.frame cannot be changed. Only accessed or removed.
X[1, ,drop=FALSE]
```

---

# pkgs *Miscellaneous package functions*

---

## Description

The `pkgs %installed in% lib.loc` operator checks if one or more package(s) `pkgs` exist(s) in library location `lib.loc`, and does so WITHOUT attaching or even loading the package(s).
Moreover, this operator forces the user to make it syntactically explicit where one is looking for installed R package(s).

The `pkg_get_deps()` function gets the dependencies of a package from the Description file. It works on non-CRAN packages also.

The `help.import()` function finds the help file for exposed infix operators and functions in an alias object.

The `pkg_lsf(package, ...)` function gets a list of exported functions/operators from a package.
One handy use for this function is to, for example, globally attach all infix operators from a function using `library`, like so:
`library(packagename, include.only = pkg_lsf("packagename", type="inops"))`

## Usage

```
pkgs %installed in% lib.loc

pkg_get_deps(
  package,
  lib.loc = .libPaths(),
```

```
  deps_type = c("LinkingTo", "Depends", "Imports"),
  base = FALSE,
  recom = FALSE,
  rstudioapi = FALSE
)

help.import(..., i, alias)

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

## Arguments

| | |
|---|---|
| pkgs | a single string, or character vector, with the package name(s). |
| lib.loc | character vector specifying library search path (the location of R library trees to search through).<br>This is usually `.libPaths()`.<br>See also loadNamespace. |
| package | a single string giving the package name. |
| deps_type | a character vector, giving the dependency types to be used.<br>Defaults to `c("LinkingTo", "Depends", "Imports")`.<br>The order of the character vector given in `deps_type` affects the order of the returned character vector; see Details sections. |
| base | logical, indicating whether base/core R should be included (`TRUE`), or not included (`FALSE`; the default). |
| recom | logical, indicating whether the pre-installed "recommended" R packages should be included (`TRUE`), or not included (`FALSE`; the default). Note that only the recommended R packages actually installed in your system are taken into consideration. |
| rstudioapi | logical, indicating whether the `rstudioapi` R package should be included (`TRUE`), or not included (`FALSE`; the default). |
| ... | further arguments to be passed to help. |
| i | either one of the following:<br>• a function (use back-ticks when the function is an infix operator). Examples: `myfun` , `` `%operator%` `` , `myalias.$some_function` .<br>• a string giving the function name or topic (i.e. `"myfun"`, `"thistopic"`). If a string, argument `alias` must be specified also. |
| alias | the alias object as created by the import_as function. Only needs to be specified if argument `i` is a string, otherwise it is ignored. |
| type | The type of functions to list. Possibilities:<br>• `"inops"` or `"operators"`: Only infix operators.<br>• `"regfuns"`: Only regular functions (thus excluding infix operators).<br>• `"all"`: All functions, both regular functions and infix operators. |

## Details

For `help.import(...)`:
Do not use the `topic` / `package` and `i` / `alias` arguments together. It's either one set or the other.

For pkg_get_deps():
If using the pkgs_get_deps() function to fill in the dependencies argument of the import_as function, one may want to know the how character vector returned by pkgs_get_deps() is ordered.
The order is determined as follows.
For each string in argument deps_type, the package names in the corresponding field of the Description file are extracted, in the order as they appear in that field.
The order given in argument deps_type also affects the order of the returned character vector:
The default,
c("LinkingTo", "Depends", "Imports"),
means the package names are extracted from the fields in the following order:

1. "LinkingTo";
2. "Depends";
3. "Imports".

The unique (thus non-repeating) package names are then returned to the user.

## Value

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.

For pkg_get_deps():
A character vector of unique dependencies.

For pkg_lsf():
Returns a character vector of function and/or operator names.

For help.import():
Opens the appropriate help page.

## References

https://stackoverflow.com/questions/30223957/elegantly-extract-r-package-dependencies-of-a-package-not-listed-on-cran

## See Also

tinyoperations_import()

## Examples

```
## Not run:
pkgs <- c(unlist(tools::package_dependencies("devtools")), "devtools")
pkgs %installed in% .libPaths()
pkg_lsf("devtools", "all")
import_as(m., "magrittr")
import_inops("magrittr")
help.import(i=m.$add)
help.import(i=`%>%`)
help.import(i="add", alias=m.)
```

```
## End(Not run)
```

---

`source_module` *Additional module import management*

---

## Description

The `alias %@source% 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% 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% 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.<br>Syntactically invalid names are not allowed for the alias name. |
| `lst` | a named list, giving the arguments to be passed to the source function.<br>For example: `alias %@source% list(file="mydir/myscript.R")`<br>The `local` argument should not be included in the list. |
| `...` | arguments to be passed to the source function, such as the `file` argument.<br>The `local` argument should not be included. |

## Value

For the `alias %@source% list(file=...)` operator:
The variable named as the `alias` will be created (if it did not already exist) in the current environment (like the Global environment, or the environment within a function), and will contain all objects from the sourced script.
To use, for example, function "some_function()" from alias "alias.", use:
`alias.$some_function()`

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).

## See Also

tinyoperations_import, base::source()

## Examples

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

## End(Not run)

exprs <- expression({
helloworld = function()print("helloworld")
goodbyeworld <- function() print("goodbye world")
`%s+test%` <- function(x,y) stringi::`%s+%`(x,y)
`%s*test%` <- function(x,y) stringi::`%s*%`(x,y)
})

myalias. %@source% list(exprs=exprs)

myalias.$helloworld()

temp.fun <- function(){
  source_inops(exprs=exprs) # places the function inside the function environment
  ls() # list all objects residing within the function definition
}
temp.fun()
```

---

# strcut_loc *Cut strings*

---

## Description

The strcut_loc() function cuts every string in a character vector around a location range loc, such that every string is cut into the following parts:

- the sub-string **before** loc;
- the sub-string at loc itself;
- the sub-string **after** loc.

The location range loc would usually be matrix with 2 columns, giving the start and end points of some pattern match.
When for some row i, loc[i, ] is c(NA, NA), loc[i, ] is translated to c(1, nc[i]), where nc[i] is the number of characters of str[i]

The strcut_brk() function, (a wrapper around stri_split_boundaries), cuts every string into individual text breaks (like character, word, line, or sentence boundaries).

The main difference between the strcut_ - functions and stri_split / strsplit, is that the latter generally removes the delimiter patterns in a string when cutting, while the strcut_-functions do not

attempt to remove parts of the string by default, they only attempt to cut the strings into separate pieces. Moreover, the `strcut_` - functions always return a matrix, not a list.

## Usage

```
strcut_loc(str, loc)

strcut_brk(str, brk = "chr", ...)
```

## Arguments

| | |
|---|---|
| `str` | a string or character vector. |
| `loc` | Either one of the following:<br>• the result from the stri_locate_ith function.<br>• a matrix of 2 integer columns, with `nrow(loc)==length(str)`, giving the location range of the middle part.<br>• a vector of length 2, giving the location range of the middle part. |
| `brk` | a single string, giving one of the following:<br>• `"chr"`: attempts to split string into individual characters.<br>• `"line"`: attempts to split string into individual lines (NOTE: this is somewhat locale dependent).<br>• `"word"`: attempts to split string into individual words (NOTE: this is highly locale dependent!).<br>• `"sentence"`: attempts to split string into individual sentences (NOTE: this is highly locale dependent!).<br>For information on the boundary rules and definitions, please see:<br>The ICU User Guide on Boundary Analysis<br>(`https://unicode-org.github.io/icu/userguide/boundaryanalysis/`) |
| `...` | additional settings for stri_opts_brkiter |

## Value

For the `strcut_loc()` function:
A character matrix with `length(str)` rows and 3 columns:

- the first column contains the sub-strings **before** `loc`;
- the second column contains the sub_strings at `loc`;
- the third and last column contains the sub-strings **after** `loc`.

For the `strcut_brk()` function:
A character matrix with `length(str)` rows and a number of columns equal to the maximum number of pieces `str` was cut in.

## See Also

`tinyoperations_stringi()`

## Examples

```
x <- rep(paste0(1:10, collapse=""), 10)
print(x)
loc <- stri_locate_ith(x, 1:10, fixed = as.character(1:10))
strcut_loc(x, loc)
strcut_loc(x, c(5,5))

test <- "The\u00a0above-mentioned    features are very useful. " %s+%
"Spam, spam, eggs, bacon, and spam. 123 456 789"
strcut_brk(test, "line")
strcut_brk(test, "word")
strcut_brk(test, "sentence")
strcut_brk(test, "chr")
```

---

# 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.<br>• 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.<br>• 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.

## See Also

`tinyoperations_stringi()`

## Examples

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

x <- c("Hello world", "Goodbye world")
print(x)
mat <- strcut_brk(x, "chr")
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 <- strcut_brk(x, "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 <- strcut_brk(x, "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 $i$ˆ$th$ 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.<br>Positive numbers are counting from the left. Negative numbers are counting from the right. I.e.:<br>• `stri_locate_ith(str, i=1, ...)` gives the position (range) of the first occurrence of a pattern.<br>• `stri_locate_ith(str, i=-1, ...)` gives the position (range) of the last occurrence of a pattern.<br>• `stri_locate_ith(str, i=2, ...)` gives the position (range) of the second occurrence of a pattern.<br>• `stri_locate_ith(str, i=-2, ...)` gives the position (range) of the second-last occurrence of a pattern.<br>If `abs(i)` is larger than the number of instances, the first (if `i < 0`) or last (if `i > 0`) instance will be given.<br>For example: suppose a string has `3` instances of some pattern;<br>then if `i >= 3` the third instance will be located,<br>and if `i <= -3` the first instance will be located. |
| `...` | more arguments to be supplied to stri_locate and stri_count.<br>Do not supply the arguments `omit_no_match`, `get_length`, or `pattern`, as they are already specified internally. Supplying these arguments anyway will result in an error. |
| `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, two `NA`s if no matches are found, and also two `NA`s if `str` is `NA`.

## See Also

`tinyoperations_stringi()`

## 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` | either a list as returned by 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

Be aware of the precedence here!
These are not regular arithmetic; these are functions. Functions come before all arithmetic in R.
For example, the following code:
`"a" %s*% 3^2`
is interpreted as: `("a" %s*% 3)^2`
which of course gives an error, since you cannot square a character.
Therefore, put brackets around the right hand side expression when using chaining arithmetic, like so:
`"a" %s*% (3^2)`

## 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.

## See Also

tinyoperations_stringi()

## 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 string %ss% ind 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 character vector x.

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

## Usage

```
string %ss% ind

x %sget% ss

x %strim% ss
```

## Arguments

| | |
|---|---|
| string | a single string. |
| ind | a numeric vector giving the subset indices. |
| x | a character vector. |
| 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 and non-missing integers, or be coerce-able to such integers. (thus negative integers, and missing values are not allowed; decimal numbers will be converted to integers).<br>The first element/column of ss gives the number of characters counting from the left side to be extracted/removed from x.<br>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 x %sget% ss operator gives a certain number of the first and last characters of character vector x.

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

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

## See Also

tinyoperations_stringi()

## 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

"hello" %ss% 5:1
```

---

subset_if *The subset_if operators and the unreal in place modifier*

---

## Description

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

The `x %unreal =% repl` operator modifies all unreal (`NA`, `NaN`, `Inf`, `-Inf`) values of `x` with replacement value `repl`.
Thus this, `x %unreal =% repl`,
is the same as,
`x[is.na(x)|is.nan(x)|is.infinite(x)] <- repl`

## Usage

```
x %[if]% cond

x %[!if]% cond

x %unreal =% repl
```

## Arguments

| | |
|---|---|
| x | a vector, matrix, or array. |
| cond | a (possibly anonymous) function that returns a `logical` vector of the same length/dimensions as `x`.<br>For example: `\(x)x>0`.<br>#' |

repl the replacement value.

## Value

For the **subset-if** operators:
The subset_if - operators all return a vector with the selected elements.

For the x %unreal =% repl operator:
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.

## See Also

tinyoperations_dry()

## Examples

```
x <- c(-10:9, NA, NA)
object_with_very_long_name <- matrix(x, ncol=2)
print(object_with_very_long_name)
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)
```

---

# s_pattern *Pattern specifications for string related infix operators.*

---

## 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.

To keep your code more compact when working with infix operators, 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`.

## See Also

`tinyoperations_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.
```

---

# tinyoperations_dry *The tinyoperations "DRY" functionality*

---

### Description

"Don't Repeat Yourself", sometimes abbreviated as "DRY", is the coding principle not to write unnecessarily repetitive code. To help you in that effort, the `tinyoperations` R package introduces a few functions:

- The transform_if function
- The subset_if operators and the in-place unreal modifier operator.
- The generalized in-place (mathematical) modification operator.
- Atomic type casting without stripping attributes.
- Infix operators for in-place modifying string arithmetic.
- Infix operators for in-place modifying string sub-setting.

Please refer to the Read-Me file on the GitHub main page of this page for more information.
See: `https://github.com/tony-aw/tinyoperations`.

### Usage

```
tinyoperations_dry()
```

### See Also

`tinyoperations_help()`

---

## `tinyoperations_help` *The tinyoperations help page*

---

### Description

Welcome to the `tinyoperations` introduction help page!

The `tinyoperations` R-package adds some infix operators, and a few functions.
It primarily focuses on 4 things:

(1) Safer decimal numbers ("double") truth testing (see %d==%).
(2) Extending the string manipulation capabilities of the `stringi` R package, see tinyoperations_stringi.
(3) Reducing repetitive code; see tinyoperations_dry.
(4) A new package and module import system, that combines the benefits of aliasing a package with the benefits of attaching a package, see tinyoperations_import

And some miscellaneous functionality; see tinyoperations_misc.

The `tinyoperations` R-package has only one dependency, namely `stringi`. Most functions in this R-package are fully vectorized and have been optimized for optimal speed and performance.

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

### Usage

```
tinyoperations_help()
```

## tinyoperations_import *The tinyoperations import system*

### Description

The tinyoperations R package introduces a new package import system. This system attempts to combine the benefits of aliasing a package, with the benefits of attaching a package.

The main part of the import system is implemented in the 3 import_ - functions:

- import_as: Allow a main package + its foreign exports + its dependencies + its enhances + its extensions to be loaded under a single alias.
- import_inops: Allow exposing infix operators to the current environment.
- import_data: Allow assigning a data set from a package directly to a variable.

The above functionality is also extended to work on sourced modules, see source_module.

There are also some additional helper functions for the package import system, see pkgs.

Please refer to the Read-Me file on the GitHub main page of this page for more information.
See: https://github.com/tony-aw/tinyoperations.

### Usage

```
tinyoperations_import()
```

### See Also

tinyoperations_help

## tinyoperations_misc *The tinyoperations miscellaneous functionality*

### Description

Some additional functions provided by the tinyoperations R package:

- Infix logical operators for exclusive-or, not-and, not-in, number-type, and string-type.
- restrict re-assignment of "T" and "F".
- create unchangeable CONSTANT.
- Infix operators for row- and column-wise re-ordering of matrices.

Please refer to the Read-Me file on the GitHub main page of this page for more information.
See: https://github.com/tony-aw/tinyoperations.

### Usage

```
tinyoperations_misc()
```

## See Also

tinyoperations_help()

---

`tinyoperations_stringi` *The tinyoperations expansion of the 'stringi' R package*

---

## Description

The `tinyoperations` R package adds some functions and operators to extend the functionality of the `stringi` R package:

- Infix operators for string arithmetic.
- Infix operators for string sub-setting.
- Infix operators for In-place modifying string arithmetic.
- Infix operators for In-place modifying string sub-setting.
- The s_pattern helper function for string infix operators.
- Infix operators for row- and column-wise re-ordering of matrices.
- The `tinyoperations` 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 strcut_-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 the functions are written in R, they have been aggressively optimized to be in the same order of speed as the other `stringi` functions.

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

## Usage

```
tinyoperations_stringi()
```

## See Also

tinyoperations_help()

transform_if *The transform_if function*

## Description

The transform_if() function is alternative form of ifelse. The transform_if() function transforms an object x, based on logical (TRUE, FALSE, NA) condition cond(x), such that:

- For every value where cond(x)==TRUE, function yes(x) is run.
- For every value where cond(x)==FALSE, function no(x) is run.
- For every value where cond(x)==NA, function other(x) is run.

## Usage

```
transform_if(
  x,
  cond,
  yes = function(x) x,
  no = function(x) x,
  other = function(x) return(NA),
  text = NULL
)
```

## Arguments

| | |
|---|---|
| x | a vector, matrix, or array. |
| cond | a (possibly anonymous) function that returns a logical vector of the same length/dimensions as x.<br>For example: \(x)x>0. |
| yes | the (possibly anonymous) transformation function to use when cond(x)==TRUE.<br>For example: log.<br>If this is not specified, yes defaults to function(x)x. |
| no | the (possibly anonymous) transformation function to use when cond(x)==FALSE.<br>For example: log.<br>If this is not specified, no defaults to function(x)x. |
| other | the (possibly anonymous) transformation function to use when cond(x) returns NA.<br>If this is not specified, other defaults to function(x)return(NA).<br>Note that other(x) is run when cond(x) is NA, not necessarily when x itself is NA! |
| text | a single string, to be used instead of the arguments cond, yes, no, other, in the form of<br>"x ; cond ; yes ; no ; other", with "x" being the declared variable.<br>If this string is given, cond, yes, no, other will be ignored.<br>The single string in question should have the following properties: |

- It should consist of 5 pieces of text, separated by semicolons (;).
- The first piece should be a single name, declaring the name of the variable.
- The second, third, fourth, and fifth pieces of text should give the expressions describing the functions to use for `cond`, `yes`, `no`, `other`, respectively, in exactly that order.
- Suppose the declared variable is named `x`. Then, each expression in text pieces 2 to 5 will be translated as:
  `function(x) text`.
  Therefore, a function like `log` must be written in the text as `"log(x)"`, not just `"log"`.
- All variables used on pieces 2 to 5 that do no match the declared variable from the first piece, are taken from the environment from which `transform_if()` was called.
- ALL 5 pieces of text are mandatory; any missing piece results in an error.
- Thus the following functions (with the declared variable being `x`),
  `cond=\(x)x > y`, `yes=log`, `no=\(x)x^2`, `other=\(x)-1000`,
  can be expressed in a single string as:
  `"x; x > y; log(x) ; x^2 ; -1000"`

## Details

Be careful with coercion! For example the following code:

```
x <- c("a", "b")
transform_if(x, \(x)x=="a", as.numeric, as.logical)
```

returns:

```
[1] NA NA
```

due to the same character vector being given 2 incompatible classes.

Using `cond`, `yes`, `no`, `other` directly is faster than using the `text` argument (because the text must first be translated into functions). Thus if speed is of primary interest, don't use the text approach.

## Value

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

## See Also

`tinyoperations_dry()`

## Examples

```
x <- c(-10:9, NA, NA)
object_with_very_long_name <- matrix(x, ncol=2)
print(object_with_very_long_name)
y <- 0
z <- 1000
object_with_very_long_name |> transform_if(\(x)x>y, log, \(x)x^2, \(x)-z)
object_with_very_long_name |> transform_if(text = "x ; x>y ; log(x) ; x^2 ; -z")
```

# Index