Package 'tinycodet'

October 8, 2023

Title A Few Functions to Help in your Coding Etiquette

Version 0.0.0.9

Maintainer Tony Wilkes <tony_a_wilkes@outlook.com>

Description Adds a few functions to help in your coding etiquette.
'tinycodet' primarily focuses on 4 aspects.
1) Safer decimal (in)equality testing, safer atomic conversions, and other functions for safer coding.
2) A new package import system,
that attempts to combine the benefits of using a package without attaching,
with the benefits of attaching a package.
3) Extending the string manipulation capabilities of the 'stringi' R package.
4) Reducing repetitive code.
'tinycodet' has only one dependency, namely 'stringi'.
Most functions in this R-package are vectorised and optimised.

License MIT + file LICENSE

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests knitr,
rmarkdown,
tinytest,
roxygen2,
data.table,
tidytable

Depends R (>= 4.1.0)

Imports stringi (>= 1.7.12)

URL https://github.com/tony-aw/tinycodet/, https://tony-aw.github.io/tinycodet/

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

R topics documented:

`atomic_conversions` *Atomic Type Casting Without Stripping Attributes*

Description

Atomic type casting in R is generally performed using the functions as.logical, as.integer, as.double, as.character.

Converting an object between atomic types using these functions strips the object of its attributes, including attributes such as names and dimensions.

The functions provided here by the 'tinycodet' package preserve all attributes - except the "class" attribute.

The functions are as follows:

- `as_bool()`: converts object to class `logical` (`TRUE`, `FALSE`, `NA`).
- `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 = sqrt(.Machine$double.eps))
```

Arguments

x	vector, matrix, array (or a similar object where all elements share the same class).
...	further arguments passed to or from other methods.
tol	numeric, giving the tolerance.

Value

The converted object.

See Also

tinycodet_safer()

Examples

```
x <- c(rep(0, 2), seq(0, 2.5, by=0.5)) |> matrix(ncol=2)
colnames(x) <- c("one", "two")
attr(x, "test") <- "test"
print(x)

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

# is_wholenumber:
is_wholenumber(1:10 + c(0, 0.1))
```

---

decimal_truth *Safer Decimal Number (In)Equality Testing Operators*

Description

The %d==%, %d!=% %d<%, %d>%, %d<=%, %d>=% (in)equality operators 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' and 'Python'.
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
```

Arguments

x, y	numeric vectors, matrices, or arrays.
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;

Value

A logical vector with the same dimensions as x, indicating the result of the element by element comparison.

See Also

tinycodet_safer()

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 <- cbind(x-0.1, x+0.1)
x %d{}% bnd
x %d!{}% bnd

# These operators work for integers also:
x <- 1L:5L
y <- 1L:5L
x %d==% y
x %d!=% y
x %d<% y
x %d>% y
x %d<=% y
x %d>=% y

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

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

`import_as` *Load R-package and its Re-exports and/or its (Reverse) Dependencies Under a Single Alias*

Description

The `import_as()` function imports the namespace of an R-package, and optionally also its re-exports, dependencies, and extensions, all 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,
  re_exports = TRUE,
  dependencies = NULL,
  extensions = NULL,
  lib.loc = .libPaths(),
  loadorder = c("dependencies", "main_package", "extensions")
)
```

Arguments

`alias`	a syntactically valid non-hidden name giving the alias object where the package(s) are to be loaded into. This name can be given either as a single string (i.e. `"alias."`), or as a one-sided formula with a single term (i.e. `~ alias.`).
`main_package`	a single string, giving the name of the main package to load under the given alias.
`re_exports`	`TRUE` or `FALSE`. • If `re_exports = TRUE` the re-exports from the `main_package` are added to the alias together with the main package. This is the default, as it is analogous to the behaviour of base R's :: operator. • If `re_exports = FALSE`, these re-exports are not added together with the main package. The user can still load the packages from which the re-exported functions came from, by specifying them in the `dependencies` argument.
`dependencies`	an optional character vector, giving the names of the dependencies of the `main_package` to be loaded also under the alias. Defaults to `NULL`, which means no dependencies are loaded. See pkg_get_deps to quickly get dependencies from a package.
`extensions`	an optional character vector, giving the names of the extensions of the `main_package` to be loaded also under the alias. 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).
The `lib.loc` argument would usually be `.libPaths()`.
See also loadNamespace.

`loadorder` the character vector
`c("dependencies", "main_package", "extensions")`,
or some re-ordering of this character vector, giving the relative load order of the groups of packages.
See Details section for more information.

Details

Expanded Definitions of Some Arguments

- "Re-exports" are functions that are defined in the dependencies of the `main_package`, but are re-exported in the namespace of the `main_package`.
- "Dependencies" are here defined as any R-package appearing in the "Depends", "Imports", or "LinkingTo" fields of the Description file of the `main_package`. So no recursive dependencies.
- "Extensions" are here defined as direct reverse-depends or direct reverse-imports. It does not matter if these are CRAN or non-CRAN packages. However, the intended meaning of an extension is not merely a reverse dependency, but a package that actually extends the functionality of the `main_package`.

Why Aliasing Multiple Packages is Useful

To use an R-package with its extension packages or dependencies, whilst avoiding the disadvantages of attaching a package (see tinycodet_import), one would traditionally use the :: operator like so:

```
main_package::some_function1()
extension1::some_function2()
extension2::some_function3()
```

This becomes cumbersome as more packages are needed and/or as the package name(s) become longer.
The `import_as()` function avoids this issue by allowing multiple **related** packages to be loaded under a single alias, allowing one to code like this:

```
import_as(
   ~ alias., "main_package",
   extensions = c("extension1", "extension2"),
   lib.loc = .libPaths()
)
alias.$some_function1()
alias.$some_function2()
alias.$some_function3()
```

Thus loading a package, or multiple directly related packages, under a single alias, which `import_as()` provides, avoids the above issues. Loading (a) package(s) under an alias is known as "aliasing" (a) package(s).

Notice that the `import_as()` function has the `lib.loc` argument, allowing to specify the library path, which the :: operator does not directly provide.

Alias Naming Recommendation
To keep package alias object names easily distinguishable from other objects that can also be subset with the $ operator, I recommend ending (not starting!) all alias names with a dot (`.`) or underscore (_).

Regarding the Load Order
The order of the character vector given in the `dependencies` and `extensions` arguments matters. If multiple packages share objects with the same name, the objects of the package named last will overwrite those of the earlier named packages.

The `loadorder` argument defaults to the character vector
`c("dependencies", "main_package", "extensions")`,
which is the recommended setting.
This setting 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 re-exports (if `re_exports = TRUE`).
3. The extensions, **in the order specified by the** `extensions` **argument**.

Other Details
The `import_as()` function does not support loading base/core R under an alias.

Value

A locked environment object, similar to the output of loadNamespace, with the name as specified in the `alias` argument, will be created in the current environment (like the global environment, or the environment within a function).
This locked environment contains the **exported** functions from the specified packages.

To use, for example, function "some_function()" from alias "alias.", use:
`alias.$some_function()`
To see the special attributes of this alias object, use attr.import.
To "unload" the package alias object, simply remove it (i.e. `rm(list="alias.")`).

See Also

`tinycodet_import()`

Examples

```
import_as( # this creates the 'tdt.' object
  "tdt.", "data.table", extensions = "tidytable"
)
# same as:
import_as(
  ~ tdt., "data.table", extensions = "tidytable"
)
```

`import_data` *Directly Return a Data-set From a Package*

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

Arguments

`package`	a single string, giving the name of the R-package.
`dataname`	a single string, giving the name of the data set.
`lib.loc`	character vector specifying library search path (the location of R library trees to search through). The `lib.loc` argument would usually be `.libPaths()`. See also loadNamespace.

Value

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

See Also

`tinycodet_import()`

Examples

```
d <- import_data("datasets", "cars")
head(d)
```

`import_inops` *(Un)Expose Infix Operators From Package Namespace in the Current Environment*

Description

`import_inops(expose=...)` exposes infix operators specified in a package or an alias object to the current environment (like the global environment or the environment within a function).

`import_inops(unexpose=...)` unexposes (i.e. removes) the infix operators specified in a package or an alias object from the current environment (like the global environment or the environment within a function).
Note that in this case only infix operators exposed by the 'tinycodet' import system will be removed from the current environment; "regular" (i.e. user-specified) infix operators will not be touched.

Usage

```
import_inops(expose = NULL, unexpose = NULL, lib.loc = .libPaths(), ...)
```

Arguments

`expose, unexpose`
: either one of the following:
  - an alias object as produced by the import_as function.
  - a string giving the package name.

`lib.loc`
: character vector specifying library search path (the location of R library trees to search through).
Only used when supplying a string to `expose` / `unexpose`, and ignored when supplying an alias object to `expose` / `unexpose` (the library is path already stored inside the alias object).
The `lib.loc` argument would usually be `.libPaths()`.
See also loadNamespace.

`...`
: additional arguments, only relevant if the `expose` argument is used.
See import_inops.control.

Details

Why Exposing Infix Operators Is Useful

To use a function from an R-package, while avoiding the disadvantages of attaching a package (see tinycodet_import), one would traditionally use the :: operator like so:

```
packagename::function_name()
```

This is, however, cumbersome with infix operators, as it forces one to code like this:

```
packagename::`%op%`(x,y)
```

Exposing infix operators to the current environment, using the `import_inops()` function, allows one to use infix operators without using cumbersome code, and without having to attach the infix operators globally.

Other Details

The `import_inops()` function does not support overloading base/core R operators.

When using `import_inops()` to remove infix operators from the current environment, it will use the attributes of those operators to determine if the infix operator came from the 'tinycodet' import system or not. Only infix operators exposed by the 'tinycodet' import system will be removed.

Value

If using argument `expose`:
The infix operators specified in the given package or alias will be placed in the current environment (like the Global environment, or the environment within a function).

If using argument `unexpose`:
The infix operators specified in the given package or alias, exposed by `import_inops()`, will be removed from the current environment (like the Global environment, or the environment within a function).
If such infix operators could not be found, this function simply returns `NULL`.

See Also

`tinycodet_import()`, `import_inops.control()`, `report_inops()`

Examples

```
import_inops(expose = "stringi") # expose infix operators from package
import_inops(unexpose = "stringi") # remove the exposed infix operators from environment

import_as(~ stri., "stringi")
import_inops(expose = stri.) # expose infix operators from alias
import_inops(unexpose = stri.) # unexposed infix operators from current environment


# additional arguments (only used when exposing, not unexposing):
import_inops(expose = "stringi", exclude = "%s==%")
import_inops(unexpose = "stringi")
import_inops(expose = "stringi", overwrite = FALSE)
import_inops(unexpose = "stringi")

import_as(~ stri., "stringi")
import_inops(expose = stri., include.only = "%s==%")
import_inops(unexpose = stri.)
import_inops(expose = stri., overwrite = FALSE)
import_inops(unexpose = stri.)
```

`import_inops.control` *import_inops.control*

Description

Additional arguments to control exposing infix operators in the import_inops function.

Usage

```
import_inops.control(
  exclude = NULL,
  include.only = NULL,
  overwrite = TRUE,
  inherits = FALSE
)
```

Arguments

`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; indicating whether enclosed environments, especially package namespaces, should also be taken into account (`TRUE`), or not (`FALSE`). Defaults to `FALSE`. See also exists.

Details

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

Value

This function is used internally in the import_inops function.

See Also

import_inops(), tinycodet_import()

Examples

```
# additional arguments (only used when exposing, not unexposing):
import_as(~ stri., "stringi")
import_inops(expose = stri., include.only = "%s==%")
import_inops(unexpose = stri.)
import_inops(expose = "stringi", exclude = "%s==%")
import_inops(unexpose = "stringi")
import_inops(expose = stri., overwrite = FALSE)
import_inops(unexpose = stri.)
import_inops(expose = "stringi", overwrite = FALSE)
import_inops(unexpose = "stringi")
```

---

import_LL *Miscellaneous import_ - Functions*

Description

The `import_LL()` function places specific functions from a package in the current environment, and also locks (see lockBinding) the specified functions to prevent modification.
The primary use-case for this function is for loading functions inside a local environment, like the environment within a function.

The `import_int()` function directly returns an internal function from a package.
It is similar to the ::: operator, but with 2 key differences:

1. It allows the user to explicitly set a library location through the `lib.loc` argument.
2. It only searches internal functions, not exported ones. This makes it clearer in your code that you're using an internal function, instead of making it ambiguous.

Usage

```
import_LL(package, selection, lib.loc = .libPaths())

import_int(form, lib.loc = .libPaths())
```

Arguments

`package`	a single string, giving the name of the package to take functions from.
`selection`	a character vector of function names (both regular functions and infix operators). Internal functions or re-exported functions are not supported.

`lib.loc`	character vector specifying library search path (the location of R library trees to search through). The `lib.loc` argument would usually be `.libPaths()`. See also loadNamespace.
`form`	a two-sided formula, with one term on each side. The term on the left hand side should give a single package name. The term on the right hand side should give a single internal function. Example: `package_name ~ function_name`

Details

Regarding the Locks in `import_LL()`
The import_as function returns a locked environment, just like loadNamespace, thus protecting the functions from accidental modification or re-assignment.
The import_inops function returns infix operators, and though these are not locked, one needs to surround infix operators by back ticks to re-assign or modify them, which is unlikely to happen on accident.
The `import_LL()` function, however, returns "loose" functions. And these functions (unless they are infix operators) do not have the protection due to a locked environment or due to the syntax.
Therefore, to ensure safety from (accidental) modification or re-assignment, the `import_LL()` function locks these functions (see lockBinding). For consistency, infix operators exposed by `import_LL()` are also locked.

Other Details
The `import_LL()` and `import_int()` functions do not support importing functions from base/core R.

Value

For `import_LL()`:
The specified functions will be placed in the current environment (like the global environment, or the environment within a function), and locked.
To "unload" or overwrite the functions, simply remove them; i.e.:
`rm(list=c("some_function1", "some_function2"))`.

For `import_int()`:
The function itself is returned directly.
So one can assign the function directly to some variable, like so:
`myfun <- import_int(...)`
or use it directly without re-assignment like so:
`import_int(...)(...)`

See Also

`tinycodet_import()`

Examples

```
# Using import_LL ====
import_LL(
```

```
  "stringi", "stri_sub"
)
# the stri_sub() function now cannot be modified, only used or removed, because it's locked:
bindingIsLocked("stri_sub", environment()) # TRUE


# Using internal function ====
# Through re-assignment:
fun <- import_int(tinycodet ~ .internal_paste, .libPaths())
fun("hello", "world")
# Or using directly:
import_int(
  tinycodet ~ .internal_paste, .libPaths()
)("hello", "world")
```

`inplace` *General In-place Modifier Operator*

Description

The `x %:=% f` operator performs 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
```

Usage

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

Arguments

`x`	a variable.
`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 modifier, and thus modifies the object directly.

See Also

`tinycodet_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)
```

---

lock *Lock T, Lock F, or Create Locked Constants*

Description

One can re-assign the values T and F. One can even run things like T <- FALSE and F <- TRUE !
The lock_TF() function locks the T and F values and sets them to TRUE and FALSE, respectively, 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(env)

X %<-c% A
```

Arguments

env	an optional environment to give, determining in which environment T and F should be locked. When not specified, the current environment (like the global environment, or the environment within a function) is used.
X	a syntactically valid unquoted name of the object to be created.
A	any kind of object to be assigned to X.

Details

Note that following statement

```
x %<-c% 2+2
print(x)
```

returns

```
[1] 2
```

due to R's precedence rules. Therefore, in such cases, the right hand side of `X %<-c% A` need to be surrounded with brackets. I.e.:

```
x %<-c% (2+2)
```

Value

For `lock_TF()`:
Two `constants`, namely `T` and `F`, set to `TRUE` and `FALSE` respectively, are created in the specified or else 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

`tinycodet_safer()`

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]
```

---

logic_ops *Additional 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 checks for every value of numeric vector `n` if it can be considered a number belonging to type `numtype`.

The `s %=strtype% strtype` operator checks for every value of character vector `s` if it can seen as a certain `strtype`.

Usage

```
x %xor% y

x %n&% y

x %out% y

x %?=% y

n %=numtype% numtype

s %=strtype% strtype
```

Arguments

x, y	see Logic.
n	a numeric vector.
numtype	a single string giving the numeric type to be checked. See Details section for supported types.
s	a character vector.
strtype	a single string giving the string type to be checked. See Details section for supported types.

Details

For argument numtype, 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);
- "I": Integers;
- "odd": odd integers;
- "even": even integers;
- "R": Real numbers;
- "unreal": infinity, NA, or NaN;

For argument strtype, 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.

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"]
```

`matrix_ops` *Row- or 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 of x, 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 of x, 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

tinycodet_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 # randomise 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 # randomise shuffle every column independently
```

---

pkgs *Miscellaneous Package Related Functions*

Description

The pkgs %installed in% lib.loc operator checks if one or more package(s) pkgs exist(s) in library location lib.loc, without loading the package(s).
The syntax of this operator forces the user to make it syntactically explicit where to look 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 pkg_lsf() 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
)

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

Arguments

pkgs	a character vector with the package name(s).
lib.loc	character vector specifying library search path (the location of R library trees to search through). The lib.loc argument would usually be .libPaths(). See also loadNamespace.
package	a single string giving the package name.
deps_type	a character vector, giving the dependency types to be used. Defaults to c("LinkingTo", "Depends", "Imports"). 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).
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

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 named logical vector, with the names giving the package names, and where the value TRUE indicates a package is installed, and the value 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 exported function names in the specified package.

References

O'Brien J., elegantly extract R-package dependencies of a package not listed on CRAN. *Stack Overflow*. (1 September 2023). https://stackoverflow.com/questions/30223957/elegantly-extract-r-package-d

See Also

tinycodet_import()

Examples

```
check <- "dplyr" %installed in% .libPaths()

if(check) pkgs <- pkg_get_deps("dplyr") # many dependencies
if(check) pkgs %installed in% .libPaths()
if(check) pkg_lsf("dplyr", "all")
```

---

report_inops *Report Infix Operators*

Description

The report_inops() function returns a data.frame listing the infix operators defined in the current environment (like the global environment, or the environment within a function), or a user specified environment. It also reports from which packages the infix operators came from.

Usage

```
report_inops(env)
```

Arguments

env	an optional environment to give, where the function should look for infix operators. When not specified, the current environment (like the global environment, or the environment within a function) is used.

Value

A data.frame. The first column gives the infix operator names. The second column gives the package the operator came from, or NA if it did not come from a package.

See Also

tinycodet_misc()

Examples

```
report_inops()

`%paste%` <- function(x,y)paste0(x,y)

report_inops()

import_inops("stringi")

report_inops()
```

---

source_selection *Source Specific Objects from Script*

Description

The source_selection() function is the same as base R's source function, except that it allows only placing the selected objects and functions into the current environment, instead of all objects.

The objects to be selected can be specified using any combination of the following:

- by supplying a character vector of exact object names to the select argument.
- by supplying a character vector of regex patterns to the regex argument.
- by supplying a character vector of fixed patterns to the fixed argument.

Note that the source_selection() function does NOT suppress output (i.e. plots, prints, messages) from the sourced script file.

Usage

```
source_selection(lst, select = NULL, regex = NULL, fixed = NULL)
```

Arguments

`lst`	a named list, giving the arguments to be passed to the source function. The `local` argument should not be included in the list.
`select`	a character vector, giving the exact names of the functions or objects appearing in the script, to expose to the current environment.
`regex`	a character vector of `regex` patterns (see about_search_regex). These should give regular expressions that match to the names of the functions or objects appearing in the script, to expose to the current environment. For example, to expose the following methods to the current environment, `mymethod.numeric()` and `mymethod.character()` from generic `mymethod()`, one could specify `regex = "^mymethod"`. about search: regex
`fixed`	a character vector of `fixed` patterns (see about_search_fixed). These should give fixed expressions that match to the names of the functions or objects appearing in the script, to expose to the current environment. For example, to expose the following methods to the current environment, `mymethod.numeric()` and `mymethod.character()` from generic `mymethod()`, one could specify `fixed= "mymethod"`. about search: fixed

Details

One can specify which objects to expose using arguments `select`, `regex`, or `fixed`.
The user can specify all 3 of them, but at least one of the 3 must be specified.
It is not a problem if the specifications overlap.

Value

Any specified objects will be placed in the current environment (like the Global environment, or the environment within a function).

See Also

tinycodet_misc, `base::source()`

Examples

```
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)
 mymethod <- function(x) UseMethod("mymethod", x)
 mymethod.numeric <- function(x)x * 2
 mymethod.character <- function(x)chartr(x, old = "a-zA-Z", new = "A-Za-z")
})

source_selection(list(exprs=exprs), regex = "^mymethod")
mymethod(1)
```

```
mymethod("a")


temp.fun <- function(){
  source_selection(list(exprs=exprs), regex = "^mymethod", fixed = c("%", ":="))
  ls() # list all objects residing within the function definition
}
temp.fun()

temp.fun <- function(){
  source_selection(list(exprs=exprs), select = c("helloworld", "goodbyeworld"))
  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.

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

Usage

```
strcut_loc(str, loc)

strcut_brk(str, type = "character", ...)
```

Arguments

`str`	a string or character vector.
`loc`	Either one of the following: • the result from the stri_locate_ith function. • a matrix of 2 integer columns, with `nrow(loc)==length(str)`, giving the location range of the middle part. • a vector of length 2, giving the location range of the middle part.

type — single string; either the break iterator type, one of `character`, `line_break`, `sentence`, `word`, or a custom set of ICU break iteration rules. Defaults to `"character"`.
about search: boundaries

... — additional settings for stri_opts_brkiter

Details

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.

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`, or `NA` if `loc` is `c(NA, NA)`;
- the second column contains the sub_strings at `loc`, or the uncut string if `loc` is `c(NA, NA)`;
- the third and last column contains the sub-strings **after** `loc`, or `NA` if `loc` is `c(NA, NA)`.

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.
Empty places are filled with `NA`.

See Also

`tinycodet_strings()`

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))
strcut_loc(x, c(NA, NA))

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)
```

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.

Value

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

See Also

`tinycodet_strings()`

Examples

```
#############################################################################

# Basic example

x <- matrix(letters[1:25], ncol=5, byrow = TRUE)
print(x)
stri_join_mat(x, margin=1)

x <- matrix(letters[1:25], ncol=5, byrow = FALSE)
print(x)
stri_join_mat(x, margin=2)
```

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

x <- c("Hello world", "Goodbye world")
print(x)
mat <- strcut_brk(x)
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 iˆth Pattern Occurrence or Text Boundary*

Description

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

The `stri_locate_ith_boundaries()` function locates the i^{th} text boundary (like character, word, line, or sentence boundaries).

Usage

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

stri_locate_ith_boundaries(str, i, ..., type = "character")
```

Arguments

`str`	a string or character vector.
`i`	a number, or a numeric vector of the same length as `str`. 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 >= 3` the third instance will be located, and if `i <= -3` the first instance will be located.
`...`	more arguments to be supplied to stri_locate or stri_locate_all_boundaries. 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. about search: regex about search: fixed about search: coll about search: charclass
`type`	single string; either the break iterator type, one of `character`, `line_break`, `sentence`, `word`, or a custom set of ICU break iteration rules. Defaults to `"character"`. about search: boundaries

Details

Special note regarding charclass

The `stri_locate_ith()` function is based on stri_locate_all. This generally gives results consistent with stri_locate_first or stri_locate_last, but the exception is when `charclass` pattern is used.
Where the functions stri_locate_first or stri_locate_last give the location of the first or last single character matching the `charclass` (respectively), stri_locate_all gives the start and end of **consecutive** characters.
The `stri_locate_ith()` is in this aspect more in line with stri_locate_all, as it gives the i^{th} set of consecutive characters.

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

tinycodet_strings()

Examples

```
##############################################################################

# practical example with regex & fixed ====

# input character vector:
x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)

# report ith (second and second-last) vowel locations:
p <- rep("A|E|I|O|U", 2) # vowels
loc <- stri_locate_ith(x, c(2, -2), regex=p, case_insensitive=TRUE)
print(loc)

# extract ith vowels:
extr <- stringi::stri_sub(x, from=loc)
print(extr)

# replace ith vowels with numbers:
repl <- stringi::stri_replace_all(
extr, fixed = c("a", "e", "i", "o", "u"), replacement = 1:5, vectorize_all = FALSE
)
x <- stringi::stri_sub_replace(x, loc, replacement=repl)
print(x)


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

# practical example with boundaries ====

# input character vector:
x <- c("good morning and good night",
"hello ladies and gentlemen")
print(x)

# report ith word locations:
loc <- stri_locate_ith_boundaries(x, c(-3, 3), type = "word")
print(loc)

# extract ith words:
extr <- stringi::stri_sub(x, from=loc)
print(extr)

# transform and replace words:
tf <- chartr(extr, old = "a-zA-Z", new = "A-Za-z")
x <- stringi::stri_sub_replace(x, loc, replacement=tf)
print(x)
```

```
##############################################################################

# find pattern ====

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)

# 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)


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

# Boundaries ====

test <- c(
paste0("The\u00a0above-mentioned    features are very useful. ",
      "Spam, spam, eggs, bacon, and spam. 123 456 789"),
      "good morning, good evening, and good night"
      )
loc <- stri_locate_ith_boundaries(test, i = c(1, -1), type = "word")
stringi::stri_sub(test, from=loc)
```

`str_arithmetic` *String Arithmetic Operators*

Description

String arithmetic operators.

The `x %s+% y` operator is exported from 'stringi', and concatenates character vectors `x` and `y`.

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

The `x %s*% n` operator is exported from 'stringi', and duplicates each string in `x` `n` times, and concatenates the results.

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

The `x %s//% brk` operator counts how often the text boundary specified in list `brk` occurs in each element of `x`.

The `e1 %s$% e2` operator is exported from 'stringi', and provides access to stri_sprintf in the form of an infix operator.

Usage

```
x %s-% p

x %s/% p

x %s//% brk
```

Arguments

`x`	a string or character vector.

p — either a list with 'stringi' arguments (see s_regex), or else a character vector of the same length as x with regular expressions.
about search: regex
about search: fixed
about search: coll
about search: charclass

brk — a list with arguments to be send to stri_count_boundaries.
see also stri_opts_brkiter.
about search: boundaries

Value

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

See Also

tinycodet_strings()

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 <- list(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.

test <- c(
paste0("The\u00a0above-mentioned    features are very useful. ",
"Spam, spam, eggs, bacon, and spam. 123 456 789"),
"good morning, good evening, and good night"
)
test %s//% list(type = "character")


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


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
y <- "a"
# pattern that ignores case:
p <- list(regex=rep("A|E|I|O|U", 2), case_insensitive = 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 %sget% ss` operator gets a certain number of the first and last characters of every string in character vector `x`.

The `x %strim% ss` operator trims a certain number of the first and last characters of every string in character vector `x`.

Usage

```
x %sget% ss

x %strim% ss
```

Arguments

`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). 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 `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`.

See Also

tinycodet_strings()

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
```

---

subset_if *Conditional Sub-setting and In-place Replacer of Unreal Values*

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,
`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. For example: \(x)x>0.
repl	the replacement value.

Value

For the x %[if]% cond and x %[!if]% cond 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 modifier, and thus modifies x directly. The object x is modified such that all NA, NaN, Inf, and -Inf elements are replaced with repl.

See Also

tinycodet_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_regex *Pattern Specifications for String Related Infix Operators*

Description

The %s-% and %s/% operators, as well as the string detection operators (%s{}%, %s!{}%), perform pattern matching for some purpose, where the pattern is given on the right hand side.
When a character vector or string is given on the right hand side, this is interpreted as case-sensitive `regex` patterns from 'stringi'.

Instead of giving a string or character vector of regex patterns, one can also supply a list to specify exactly how the pattern should be interpreted. The list should use the exact same argument convention as 'stringi'.

For example:

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

All arguments in the list are simply passed to the appropriate functions in 'stringi'.
For example:

```
x %s/% p
```

counts how often regular expression specified in character vector `p` occurs in `x`, whereas the following,

```
x %s/% list(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.

'tinycodet' adds some convenience functions based on the `stri_opts_` - functions in 'stringi':

- `s_regex(p, ...)` is equivalent to `list(regex = p, ...)`
- `s_fixed(p, ...)` is equivalent to `list(fixed = p, ...)`
- `s_coll(p, ...)` is equivalent to `list(coll = p, ...)`
- `s_chrcls(p, ...)` is equivalent to `list(charclass = p, ... )`

With the ellipsis (`...`) being passed to the appropriate 'stringi'-functions when it matches their arguments.

'stringi' infix operators start with "`%s`", though they all have an alias starting with "`%stri`". In analogy to that, the above functions start with "`s_`" rather than "`stri_`", as they are all meant for infix operators only.

Usage

```
s_regex(
  p,
  case_insensitive,
  comments,
```

```
  dotall,
  multiline,
  time_limit,
  stack_limit,
  ...
)

s_fixed(p, case_insensitive, overlap, ...)

s_coll(
  p,
  locale,
  strength,
  alternate_shifted,
  french,
  uppercase_first,
  case_level,
  numeric,
  normalization,
  ...
)

s_chrcls(p, ...)
```

Arguments

`p`	a character vector giving the pattern to search for. about search: regex about search: fixed about search: coll about search: charclass
`case_insensitive`	see stri_opts_regex and stri_opts_fixed.
`comments, dotall, multiline`	see stri_opts_regex.
`time_limit, stack_limit`	see stri_opts_regex.
`...`	additional arguments not part of the `stri_opts` - functions to be passed here. For example: `max_count`
`overlap`	see stri_opts_fixed.
`locale, strength, alternate_shifted`	see stri_opts_collator.
`french, normalization, numeric`	see stri_opts_collator.
`uppercase_first, case_level`	see stri_opts_collator.

Value

A list with arguments to be passed to the appropriate functions.

See Also

tinycodet_strings()

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 <- list(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)
x %s/% list(regex = rep("A|E|I|O|U", 2), case_insensitive = TRUE)
x %s/% s_regex(rep("A|E|I|O|U", 2), case_insensitive = TRUE)


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- list(fixed = c("A", "A"), case_insensitive=TRUE)
x %s{}% p
x %s!{}% p
p <- s_fixed(c("A", "A"), case_insensitive=TRUE)
x %s{}% p
x %s!{}% p
```

---

tinycodet_dry *Overview of the tinycodet "Don't Repeat Yourself" Functionality*

Description

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

- The transform_if function
- The subset_if operators and the in-place unreal modifier operator.
- The general in-place (mathematical) modification operator.

See Also

tinycodet_help()

Examples

```
object <- matrix(c(-9:8, NA, NA) , ncol=2)

# in base R:
ifelse( # repetitive, and gives unnecessary warning
  is.na(object>0), -Inf,
  ifelse(
```

```
    object>0,  log(object), object^2
  )
)
mtcars$mpg[mtcars$cyl>6] <- (mtcars$mpg[mtcars$cyl>6])^2 # long

# with tinycodet:
object |> transform_if(\(x)x>0, log, \(x)x^2, \(x) -Inf) # compact & no warning
mtcars$mpg[mtcars$cyl>6] %:=% \(x)x^2 # short
```

`tinycodet_help` *The 'tinycodet' Introduction Help Page*

Description

Welcome to the 'tinycodet' introduction help page!

'tinycodet' adds a few functions to help in your coding etiquette.
It primarily focuses on 4 things:

(1) Safer decimal (in)equality testing, safer atomic conversions, and other functions for safer coding;
see tinycodet_safer.

(2) A new package import system, that attempts to combine the benefits of using a package without attaching, with the benefits of attaching a package;
see tinycodet_import

(3) Extending the string manipulation capabilities of the 'stringi' R-package;
see tinycodet_strings.

(4) Reducing repetitive code;
see tinycodet_dry.

And some miscellaneous functionality; see tinycodet_misc.

'tinycodet' adheres to the tinyverse philosophy (not to be confused with the tidyverse). 'tinycodet' has only one dependency, namely 'stringi'. No other dependencies, thus avoiding "dependency hell". Most functions in this R-package are vectorised and optimised.

Author(s)

Maintainer: Tony Wilkes <tony_a_wilkes@outlook.com> (ORCID)

References

The badges shown in the documentation of this R-package were made using the services of: https://shields.io/

See Also

Useful links:

- 'tinycodet' GitHub page: https://github.com/tony-aw/tinycodet/
- 'tinycodet' package website: https://tony-aw.github.io/tinycodet/
- Report bugs at: https://github.com/tony-aw/tinycodet/issues/
- The fastverse, which is related to the tinyverse: https://github.com/fastverse/fastverse/

`tinycodet_import` *Overview of the 'tinycodet' Import System*

Description

The 'tinycodet' R-package introduces a new package import system.

One can **use** a package **without attaching** the package - for example by using the :: operator.
Or, one can explicitly **attach** a package - for example by using the library function.
The advantages and disadvantages of **using without attaching** a package versus **attaching** a package, at least those relevant here, are compactly presented in the following list:

(1) Prevent masking functions from other packages:
use without attach: Yes(advantage); attaching: No(disadvantage);

(2) Prevent masking core R functions:
use without attach: Yes(advantage); attaching: No(disadvantage);

(3) Clarify which function came from which package:
use without attach: Yes(advantage); attaching: No(disadvantage);

(4) Enable functions only in current/local environment instead of globally:
use without attach: Yes(advantage); attaching: No(disadvantage);

(5) Prevent namespace pollution:
use without attach: Yes(advantage); attaching: No(disadvantage);

(6) Minimise typing - especially for infix operators
(i.e. typing `package::` %op% `(x, y)` instead of `x %op% y` is cumbersome):
use without attach: No(disadvantage); attaching: Yes(advantage);

(7) Use multiple related packages, without constantly switching between package prefixes
(i.e. doing `packagename1::some_function1();`
`packagename2::some_function2();`
`packagename3::some_function3()` is chaotic and cumbersome):
use without attach: No(disadvantage); attaching: Yes(advantage);

What 'tinycodet' attempts to do with its import system, is to somewhat find the best of both worlds.
It does this by introducing the following functions:

- import_as: Load a main package, and optionally its re-exports + its dependencies + its extensions, under a single alias. This essentially combines the attaching advantage of using multiple related packages (item 7 on the list), whilst keeping most advantages of using without attaching a package.
- import_inops: Expose infix operators from a package or an alias object to the current environment. This gains the attaching advantage of less typing (item 6 on the list), whilst simultaneously avoiding the disadvantage of attaching functions from a package globally (item 4 on the list).
- import_data: Directly return a data set from a package, to allow straight-forward assignment.

Furthermore, there are two miscellaneous `import_` - functions: import_LL and import_int.
And there are also some additional helper functions for the package import system, see x.import and pkgs.

All `import_`-functions have the `lib.loc` argument to specify the library path to load packages from, thus allowing straight-forward project isolation.

See the examples section below to get an idea of how the 'tinycodet' import system works in practice. More examples can be found on the website (`https://tony-aw.github.io/tinycodet/`)

Details

When to Use or Not to Use the 'tinycodet' Import System

There is no necessity for using the 'tinycodet' import system with every single package. One can safely attach the 'stringi' package, for example, as 'stringi' uses a unique and immediately recognisable naming scheme (virtually all 'stringi' functions start with "`stri_`"), and this naming scheme does not conflict with core R, nor with most other packages.
Of course, if one wishes to use 'stringi' **only** within a specific environment, like only inside a function, it becomes advantageous to load 'stringi' using the 'tinycodet' import system (in that case the import_LL function would be most applicable).

Some Additional Comments on the 'tinycodet' Import System

- (S3) Methods will automatically be registered.
- Pronouns, such as the `.data` and `.env` pronouns from the 'rlang' package, will work without any prefixes required.

See Also

tinycodet_help

Examples

```
## Not run:

# NO packages are being attached in any of the following code

# load 'dplyr' + its re-exports + extension 'powerjoin', under alias "dpr.":
import_as(
  ~ dpr., "dplyr", re_exports = TRUE, extensions = "powerjoin"
)

# exposing infix operators from 'magrrittr' to current environment:
```

```
import_inops("magrittr")

# directly assigning dplyr's "starwars" dataset to object "d":
d <- import_data("dplyr", "starwars")

# See it in Action:
d %>% dpr.$filter(species == "Droid") %>%
  dpr.$select(name, dpr.$ends_with("color"))

male_penguins <- dpr.$tribble(
  ~name,    ~species,     ~island, ~flipper_length_mm, ~body_mass_g,
  "Giordan",    "Gentoo",    "Biscoe",               222L,        5250L,
  "Lynden",    "Adelie", "Torgersen",               190L,        3900L,
  "Reiner",    "Adelie",     "Dream",               185L,        3650L
)

female_penguins <- dpr.$tribble(
  ~name,    ~species,  ~island, ~flipper_length_mm, ~body_mass_g,
  "Alonda",    "Gentoo", "Biscoe",               211,        4500L,
  "Ola",    "Adelie",  "Dream",               190,        3600L,
  "Mishayla",    "Gentoo", "Biscoe",               215,        4750L,
)
dpr.$check_specs()

dpr.$power_inner_join(
  male_penguins[c("species", "island")],
  female_penguins[c("species", "island")]
)

mypaste <- function(x, y) {
  import_LL("stringi", selection = "stri_c")
  stringi::stri_c(x, y)
}
mypaste("hello ", "world")


## End(Not run)
```

`tinycodet_misc` *Overview of the 'tinycodet' Miscellaneous Functionality*

Description

Some additional functions provided by the 'tinycodet' R-package:

- Infix logical operators for exclusive-or, not-and, not-in, number-type, and string-type.
- Infix operators for row- and column-wise re-ordering of matrices.
- Report infix operators present in the current environment, or a specified environment.
- source_selection to source only selected objects.

See Also

`tinycodet_help()`

tinycodet_safer *Overview of the 'tinycodet' "Safer" Functionality*

Description

To help make your code safer, the 'tinycodet' R-package introduces a few functions:

- Safer decimal (in)equality testing.
- Atomic type casting without stripping attributes.
- The lock_TF function to set and lock `T` and `F` to `TRUE` and `FALSE`.
- The %<-c% operator to assign locked constants.

See Also

`tinycodet_help()`

Examples

```
x <- c(0.3, 0.6, 0.7)
y <- c(0.1*3, 0.1*6, 0.1*7)
x == y # gives FALSE, but should be TRUE
x %d==% y # here it's done correctly
```

tinycodet_strings *Overview of the 'tinycodet' Extension of 'stringi'*

Description

Virtually every programming language, even those primarily focused on mathematics, will at some point have to deal with strings. R's atomic classes basically boil down to some form of either numbers or characters. R's numerical functions are generally very fast. But R's native string functions are somewhat slow, do not have a unified naming scheme, and are not as comprehensive as R's impressive numerical functions.

The primary R-package that fixes this is 'stringi'. 'stringi' is the fastest and most comprehensive string manipulation package available at the time of writing. Many string related packages fully depend on 'stringi'. The 'stringr' package, for example, is merely a thin wrapper around 'stringi'.

As string manipulation is so important to programming languages, 'tinycodet' adds a little bit new functionality to 'stringi':

- Find i^{th} pattern occurrence (stri_locate_ith), or i^{th} text boundary (stri_locate_ith_boundaries).
- Concatenate a character matrix row- or column-wise .
- Cut strings with the strcut_-functions.
- Infix operators for string arithmetic.
- Infix operators for string sub-setting, which get or remove the first and/or last `n` characters from strings.
- Infix operators for detecting patterns.

References

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1–59, doi:10.18637/jss.v103.i02

See Also

`tinycodet_help()`, `s_regex()`

Examples

```
# character vector:
x <- c("3rd 1st 2nd", "5th 4th 6th")
print(x)

# detect if there are digits:
x %s{}% "[[:digits]]"

# cut x into matrix of individual words:
x <- strcut_brk(x, "word")

# re-order matrix using the fast %row~% operator:
mat <- stringi::stri_rank(as.vector(x)) |> matrix(ncol=ncol(x))
sorted <- x %row~% mat

# join elements of every row into a single character vector:
stri_c_mat(sorted, margin=1, sep=" ")
```

---

transform_if *The transform_if function*

Description

The `transform_if()` function transforms an object `x`, based on the logical result (`TRUE`, `FALSE`, `NA`) of condition function `cond(x)` or logical vector `cond`, such that:

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

Usage

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

Arguments

x	a vector, matrix, or array.
cond	either an object of class logical with the same length as x, or a (possibly anonymous) function that returns an object of class logical with the same length as x. For example: \(x)x>0.
yes	the (possibly anonymous) transformation function to use when function cond(x)==TRUE / logical cond==TRUE. Alternatively, one can also supply an atomic scalar. If argument yesis not specified, it defaults to \(x)x.
no	the (possibly anonymous) transformation function to use when function cond(x)==FALSE / logical cond==FALSE. Alternatively, one can also supply an atomic scalar. If argument no is not specified, it defaults to \(x)x.
other	the (possibly anonymous) transformation function to use when function cond(x) / logical cond returns NA. Alternatively, one can also supply an atomic scalar. If argument other is not specified, it defaults to NA. Note that function other(x) is run or scalar other is returned when function cond(x) or logical cond is NA, not necessarily when x itself is NA.

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.

Value

The transformed vector, matrix, or array (attributes are conserved).

See Also

tinycodet_dry()

Examples

```
x <- c(-10:9, NA, NA)
object <- matrix(x, ncol=2)
attr(object, "helloworld") <- "helloworld"
print(object)
y <- 0
```

```
z <- 1000

object |> transform_if(\(x)x>y, log, \(x)x^2, \(x)-z)
object |> transform_if(object > y, log, \(x)x^2, -z) # same as previous line
```

`x.import` *Helper functions for the tinycodet package import system*

Description

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

The `is.tinyimport()` function checks if an alias object or an exposed function is of class `tinyimport`; i.e. if it is an object produced by the import_as, import_inops, or import_LL function.

The `attr.import()` function gets one specific special attributes or all special attributes from an alias object returned by import_as.

Usage

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

is.tinyimport(x)

attr.import(alias, which = NULL)
```

Arguments

`...`	further arguments to be passed to help.
`i`	either one of the following: • a function (use back-ticks when the function is an infix operator). Examples: `myfun` , `` `%operator%` `` , `myalias.$some_function` . If a function, the `alias` argument is ignored. • 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.
`x`	the object/function to be tested.
`which`	The attributes to list. If `NULL`, all attributes will be returned. Possibilities: "pkgs", "conflicts", "versions", "args", and "ordered_object_names".

Details

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

```
import_as(~ mr., "magrittr")
import_inops(mr.)
help.import(i=mr.$add)
help.import(i=`%>%`)
help.import(i="add", alias=mr.)
help.import(topic = "%>%", package = "magrittr")
```

Value

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

For `is.tinyimport()`:
Returns `TRUE` if the function is produced by import_as, import_inops, or import_LL, and returns `FALSE` if it is not.

For `attr.import(alias, which = NULL)`:
All special attributes of the given alias object are returned as a list.

For `attr.import(alias, which = "pkgs")`:
Returns a list with 3 elements:

- packages_order: a character vector of package names, giving the packages in the order they were loaded in the alias object.
- main_package: a string giving the name of the main package. Re-exported functions, if present, are loaded together with the main package.
- re_exports.pkgs: a character vector of package names, giving the packages from which the re-exported functions in the main package were taken.

For `attr.import(alias, which = "conflicts")`:
The order in which packages are loaded in the alias object (see attribute `pkgs$packages_order`) matters: Functions from later named packages overwrite those from earlier named packages, in case of conflicts.
The "conflicts" attribute returns a data.frame showing exactly which functions overwrite functions from earlier named packages, and as such "win" the conflicts.

For `attr.import(alias, which = "versions")`:
A data.frame, giving the version of every package loaded in the alias, ignoring re-exports.

For `attr.import(alias, which = "args")`:
Returns a list of input arguments. These were the arguments supplied to import_as when the alias object in question was created.

For `attr.import(alias, which = "ordered_object_names")`:
Gives the names of the objects in the alias, in the order as they were loaded.
For conflicting objects, the last load is used for the ordering.
Note that if argument `re_exports` is `TRUE`, re-exported functions are loaded when the main package is loaded, thus changing this order slightly.

See Also

`tinycodet_import()`

Examples

```
import_as(~ to., "tinycodet")
import_inops(to.)
`%s==%` <- stringi::`%s==%`

is.tinyimport(to.) # returns TRUE
is.tinyimport(`%:=%`) # returns TRUE
is.tinyimport(`%s==%`) # returns FALSE: not imported by tinycodet import system

attr.import(to., which="conflicts")
```

`%s{}%` *'stringi' Pattern Detection Operators*

Description

The `x %s{}% p` operator checks for every string in character vector `x` if the pattern defined in `p` is present.

The `x %s!{}% p` operator checks for every string in character vector `x` if the pattern defined in `p` is NOT present.

Usage

```
x %s{}% p

x %s!{}% p
```

Arguments

`x`	a string or character vector.
`p`	either a list with 'stringi' arguments (see s_regex), or else a character vector of the same length as `x` with regular expressions. about search: regex about search: fixed about search: coll about search: charclass

Value

The `x %s{}% p` and `x %s!{}% p` operators return logical vectors, where `TRUE` indicates a pattern was found, and `FALSE` indicates a pattern was not found.

See Also

`tinycodet_strings()`

Examples

```
# simple pattern ====

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
x %s{}% "a"
x %s!{}% "a"
which(x %s{}% "a")
which(x %s!{}% "a")
x[x %s{}% "a"]
x[x %s!{}% "a"]
x[x %s{}% "a"] <- 1
x[x %s!{}% "a"] <- 1
print(x)

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
x %s{}% "1"
x %s!{}% "1"
which(x %s{}% "1")
which(x %s!{}% "1")
x[x %s{}% "1"]
x[x %s!{}% "1"]
x[x %s{}% "1"] <- "a"
x[x %s!{}% "1"] <- "a"
print(x)


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

# ignore case pattern ====


x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- list(regex = c("A", "A"), case_insensitive=TRUE)
x %s{}% p
x %s!{}% p
which(x %s{}% p)
which(x %s!{}% p)
x[x %s{}% p]
x[x %s!{}% p]
x[x %s{}% p] <- "hello"
x[x %s!{}% p] <- "hello"
print(x)
```

```
################################################################################

# multi-character pattern ====

x <- c(paste0(letters[1:13], collapse=""), paste0(letters[14:26], collapse=""))
print(x)
p <- list(regex = rep("AB", 2), case_insensitive=TRUE)
x %s{}% p
x %s!{}% p
which(x %s{}% p)
which(x %s!{}% p)
x[x %s{}% p]
x[x %s!{}% p]
x[x %s{}% p] <- "CD"
x[x %s!{}% p] <- "CD"
print(x)
```

Index