Package 'tinycodet'

September 21, 2025

Title Functions to Help in your Coding Etiquette

Version 0.5.8

Description Adds some functions to help in your coding etiquette.
'tinycodet' primarily focuses on 4 aspects.
1) Safer decimal (in)equality testing,
standard-evaluated alternatives to with() and aes(),
and other functions for safer coding.
2) A new package import system,
that attempts to combine the benefits of using a package without attaching it,
with the benefits of attaching a package.
3) Extending the string manipulation capabilities of the 'stringi' R package.
4) Reducing repetitive code.
Besides linking to 'Rcpp', 'tinycodet' has only one other dependency, namely 'stringi'.

License MIT + file LICENSE

Encoding UTF-8

LinkingTo Rcpp

RoxygenNote 7.3.2

Suggests tinytest, ggplot2, mgcv, nlme, collapse, kit, knitr,
rmarkdown, roxygen2

Depends R (>= 4.1.0)

Imports Rcpp (>= 1.0.11), 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/

Language en-gb

NeedsCompilation yes

Author Tony Wilkes [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0001-9498-8379>)

Maintainer Tony Wilkes <tony_a_wilkes@outlook.com>

Repository CRAN

Date/Publication 2025-09-21 17:30:02 UTC

Contents

`aaa0_tinycodet_help` *tinycodet: Functions to Help in your Coding Etiquette*

Description

'tinycodet' adds some functions to help in your coding etiquette.
It primarily focuses on 4 aspects:

(1) Safer decimal (in)equality testing, standard-evaluated alternatives to `with()` and `aes()`, 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 it, 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.

Please check the Change-log (see links below) regularly for updates (such as bug fixes).

'tinycodet' adheres to the tinyverse philosophy. Besides linking to 'Rcpp', 'tinycodet' only has one other dependency: 'stingi'. No other dependencies, thus avoiding "dependency hell". Most functions in this R-package are vectorized 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 main page and Read-Me: 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/
- Changelog: https://github.com/tony-aw/tinycodet/blob/main/NEWS.md/ or https://tony-aw.github.io/tinycodet/news/index.html
- The 'fastverse', which is related to the 'tinyverse': https://github.com/fastverse/fastverse/

aaa1_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.
- Standard evaluated versions of some common expression-evaluation functions: with_pro and aes_pro.
- The lock_TF function to set and lock `T` and `F` to `TRUE` and `FALSE`, respectively.
- The %<-c% operator to assign locked constants.
- safer_partialmatch to set options for safer dollar, arguments, and attribute matching.

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
```

aaa2_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: Import a main package, and optionally its re-exports + its direct dependencies + its direct 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.

The import system also includes general helper functions:

- The x.import functions: helper functions specifically for the 'tinycodet' import system.
- The pversion_ functions: check mismatch between loaded package version and package version in library path.
- The pkgs - functions: general helper functions regarding packages.

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

The 'tinycodet' import system is helpful particularly for packages that have at least one of the following properties:

- The namespace of the package(s) conflicts with other packages.
- The namespace of the package(s) conflicts with core R, or with those of recommended R packages.
- The package(s) have function names that are generic enough, such that it is not obvious which function came from which package.

See examples below.

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 a package (like 'stringi') **only** within a specific environment, it becomes advantageous to still import the package 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.
- All functions imported by the import_as, import_inops, or import_LL functions have a "package" attribute, so you will always know which function came from which package.

See Also

tinycodet_help

Examples

```
all(c("dplyr", "powerjoin", "magrittr") %installed in% .libPaths())



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

# import '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")
```

---

aaa3_tinycodet_strings

Overview of the 'tinycodet' Extension of 'stringi'

Description

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', which many, if not most, string related packages

depend on (see the list of reverse-dependencies on CRAN).
As string manipulation is important to programming languages, even those primarily focused on mathematics, 'tinycodet' adds a little bit new functionality to 'stringi'.

'tinycodet' adds the following functions to extend '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.

'tinycodet' adds the following operators, to complement the already existing 'stringi' operators:

- 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, and strfind()<- for locating/extracting/replacing found patterns.

And finally, 'tinycodet' adds the somewhat separate strcut_-functions, to cut strings into pieces without removing the delimiters.

Regarding Vector Recycling in the 'stringi'-based Functions

Generally speaking, vector recycling is supported as 'stringi' itself supports it also.
There are, however, a few exceptions.
First, matrix inputs (like in strcut_loc and string sub-setting operators) will generally not be recycled.
Second, the `i` argument in stri_locate_ith does not support vector recycling.
Scalar recycling is virtually always supported.

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_pattern

Examples

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

# detect if there are digits:
x %s{}% "\\d"

# find second last digit:
loc <- stri_locate_ith(x, i = -2, regex = "\\d")
stringi::stri_sub(x, from = loc)

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

# sort rows of matrix using the fast %row~% operator:
rank <- stringi::stri_rank(as.vector(mat)) |> matrix(ncol = ncol(mat))
sorted <- mat %row~% rank
sorted[is.na(sorted)] <- ""

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

`aaa4_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 features:

- The transform_if function.
- Operators for short-hand re-ordering matrices Row- or Column-wise.

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
```

```
  )
)

# with tinycodet:
object |> transform_if(\(x) x > 0, log, \(x) x^2, \(x) -Inf) # compact & no warning
```

`aaa5_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.
- Report infix operators present in the current environment, or a specified environment.
- source_selection to source only selected objects.

See Also

`tinycodet_help()`

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

Description

The `%d==%`, `%d!=%` `%d<%`, `%d>%`, `%d<=%`, `%d>=%` (in)equality operators perform decimal (type "double") number truth testing.
They are virtually equivalent to the regular (in)equality operators,
`==`, `!=`, `<`, `>`, `<=`, `>=`,
except for 2 aspects:

1. The decimal number (in)equality operators assume that if the absolute difference between any 2 numbers `x` and `y` is smaller than the Machine tolerance, `sqrt(.Machine$double.eps)`, then `x` and `y` should be consider to be equal.
   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`.

2. Only numeric input is allowed, so characters are not coerced to numbers.
   I.e. `1 < "a"` gives `TRUE`, whereas `1 %d<% "a"` gives an error.
   For character equality testing, see %s==% from the 'stringi' package.

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

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`.

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

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

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

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)` (or can be recycled to be `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.
`tol`	a single, strictly positive number close to zero, giving the tolerance.

Details

The operators described in this page are defined in terms of the existing base Logic operators, and should therefore be compatible with (S3) classes that have method dispatches defined for relational

operators.

Value

For the %d...% operators:
A logical vector with the same dimensions as x, indicating the result of the element by element comparison.
NOTE: Inf by Inf and -Inf by -Inf comparisons with the %d...% operators return NA.

For is_wholenumber():
A logical vector with the same dimensions as x, indicating the result of the element by element comparison.
NOTE: Inf, -Inf, NaN and NA all return NA for is_wholenumber().

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

# same as above, but here the results are correct:
x %d==% y # correct
x %d!=% y # correct
x %d<% y # correct
x %d>% y # correct
x %d<=% y # correct
x %d>=% y # correct

# check if numbers are in closed interval:
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 + 1L
x %d==% y
x %d!=% y
x %d<% y
x %d>% y
x %d<=% y
x %d>=% y

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

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

`import_as` *Import R-package, its Re-exports, Dependencies, and/or Extensions, 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, containing the exported functions from the specified packages, will be placed in the current environment.

Usage

```
import_as(
  alias,
  main_package,
  re_exports = TRUE,
  dependencies = NULL,
  extensions = NULL,
  lib.loc = .libPaths(),
  import_order = c("dependencies", "main_package", "extensions")
)
```

Arguments

`alias`	a syntactically valid name giving the alias object where the package(s) are to be imported 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 import under the given alias. Core R (i.e. "base", "stats", etc.) is not allowed.
`re_exports`	`TRUE` or `FALSE`. • If `re_exports = TRUE` the re-exports from the `main_package` (including those exported from Core R) 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 import the packages under the alias 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 imported also under the alias. Defaults to `NULL`, which means no dependencies are imported under the alias. See pkg_get_deps to quickly get dependencies from a package. Core R (i.e. "base", "stats", etc.) is not allowed.
`extensions`	an optional character vector, giving the names of the extensions of the `main_package` to be imported also under the alias. Defaults to `NULL`, which means no extensions are imported under the alias. Core R (i.e. "base", "stats", etc.) is not allowed.
`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.
`import_order`	the character vector `c("dependencies", "main_package", "extensions")`, or some re-ordering of this character vector, giving the relative import 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`.
  Unlike the `Dependencies` argument, functions from core R are included in re-exports.
- "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 reverse-dependencies that actually extend the functionality of the `main_package`. Programmatically, some package "E" is considered an extension of some "main_package", if the following is `TRUE`:
  `"main_package" %in% pkg_get_deps_minimal("E")`

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()
dependency1::some_function2()
extension1::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 imported under a single alias, allowing one to code like this:

```
import_as(
   ~ alias., "main_package",
   dependencies = "dependency1", extensions = "extension1",
   lib.loc = .libPaths()
)
alias.$some_function1()
alias.$some_function2()
alias.$some_function3()
```

Thus importing a package, or multiple directly related packages, under a single alias, which `import_as()` provides, avoids the above issues. Importing a package under an alias is referred to as "aliasing" a package.

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 (or starting, if you want to hide it from `ls()`) all alias names with a dot (`.`), or ending alias names with an underscore (`_`).

Regarding `import_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 `import_order` argument defaults to the character vector
`c("dependencies", "main_package", "extensions")`,
which is the recommended setting.
This setting results in the following importing 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

- Packages that appear in the "Suggests" or "Enhances" fields of packages are not considered dependencies or extensions.
- No more than 10 packages (ignoring re-exports) are allowed to be imported under a single alias.
- Primitive functions (see is.primitive) are not imported.

Value

A locked environment object, similar to the output of loadNamespace, with the name as specified in the `alias` argument, will be created.
This object, referred to as the "(package) alias object", will contain the exported functions from the specified package(s).
The alias object will be placed in the current environment.

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 "unimport" the package alias object, simply remove it (i.e. `rm(list = "alias.")`).

See Also

tinycodet_import

Examples

```
"data.table" %installed in% .libPaths()

import_as( # this creates the 'dt.' object
  ~ dt., "data.table"
)
```

`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_helper` *Helper Functions for the 'tinycodet' Package Import System*

Description

The `help.import()` function finds the help file for functions or topics, including exposed functions/operators as well as functions in a package alias object.

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 or all special attribute(s) 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`	an existing object (i.e. an assigned variable or a locked constant) to be tested.
`which`	The attributes to list. If `NULL`, all attributes will be returned. Possibilities: "pkgs", "conflicts", "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")
help.import("%>%", package = "magrittr") # same as previous line
```

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 imported in the alias object.
- main_package: a string giving the name of the main package. Re-exported functions, if present, are taken 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 imported 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 = "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 imported.
For conflicting objects, the last imported ones are used for the ordering.
Note that if argument `re_exports` is `TRUE`, re-exported functions are imported when the main package is imported, 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")
```

`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.

`import_inops(unexpose = ...)` "unexposes" (i.e. removes) the infix operators specified in a package or an alias object from the current environment.
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-defined) infix operators will not be touched.

To attach all infix operators from a package to the global namespace, one can use the pkg_lsf function like so:

```
y <- pkg_lsf("packagename", type = "inops")
library(packagename, include.only = y)
```

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. Core R (i.e. "base", "stats", etc.) is not allowed.

`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.

Primitive operators (see is.primitive) are not exposed.

Value

If using argument `expose`:
The infix operators specified in the given package or alias will be placed in the current environment.

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.
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. When exposing infix operators, import_inops checks if infix operators with the same names are already present in the current environment. If `inherits = FALSE`, only the current environment is checked for existing operators. If `inherits = TRUE`, enclosed environments, most notably package namespaces, are also checked for existing operators. 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 exposing functions inside a local environment.

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

1. `import_int()` includes the `lib.loc` argument.
2. `import_int()` 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. Core R (i.e. "base", "stats", etc.) is not allowed.
`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` Core R (i.e. "base", "stats", etc.) is not allowed.

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, and locked.
To unexpose 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

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



# 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")
```

---

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

Description

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 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)
```

Note that the `lock_TF()` function and `%s<-c%` operator create constants through lockBinding.
The constants are protected from modification by copy, but they are **not** protected from modification by reference (see for example `collapse::setv`).

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&% 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`.

Note that these operators strip all attributes, except dimensions.

Usage

```
x %row~% mat

x %col~% mat
```

Arguments

x	a matrix
mat	a numeric matrix with the same dimensions as x, giving the new ordering ranks for every element of matrix x.

Value

A re-ordered matrix.

See Also

tinycodet_dry

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(seq_along(x)) |> matrix(ncol = ncol(x))
x %row~% mat # randomly shuffle every row independently
x %col~% mat # randomly 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(seq_along(x)) |> matrix(ncol = ncol(x))
```

```
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 given packages (pkgs) exist in the given library paths (`lib.loc`), without loading the packages.
The syntax of this operator forces the user to make it syntactically explicit where to look for installed R-packages.
As pkgs `%installed in%` `lib.loc` does not even load a package, the user can safely use it without fearing any unwanted side-effects.

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

The `pkg_get_deps_minimal()` function is the same as `pkg_get_deps()`, except with `base, recom, rstudioapi, shared_tidy` all set to `FALSE`, and the default value for `deps_type` is c("Depends", "Imports").

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 package using `library`, like so:

```
y <- pkg_lsf("packagename", type = "inops")
library(packagename, include.only = y)
```

Usage

```
pkgs %installed in% lib.loc

pkg_get_deps(
  package,
  lib.loc = .libPaths(),
  deps_type = c("LinkingTo", "Depends", "Imports"),
  base = FALSE,
  recom = TRUE,
  rstudioapi = TRUE,
  shared_tidy = TRUE
)

pkg_get_deps_minimal(
  package,
  lib.loc = .libPaths(),
  deps_type = c("Depends", "Imports")
```

```
)

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. 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).
recom	logical, indicating whether the pre-installed 'recommended' R-packages should be included (TRUE), or not included (FALSE).
rstudioapi	logical, indicating whether the 'rstudioapi' R-package should be included (TRUE), or not included (FALSE).
shared_tidy	logical, indicating whether the shared dependencies of the 'tidyverse' should be included (TRUE), or not included (FALSE). **Details:** Some of the (often many) dependencies 'tidyverse' packages have are shared across the majority of the 'tidyverse'. The "official" list of shared dependencies in the 'tidyverse' currently is the following: 'rlang', 'lifecycle', 'cli', 'glue', and 'withr'.
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():
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:
For example, 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.
The names give the package names.
The value `TRUE` indicates a package is installed in `lib.loc`.
The value `FALSE` indicates a package is not installed in `lib.loc`.
The value `NA` indicates a package is not actually a separate package, but base/core 'R' (i.e. 'base', 'stats', etc.).

For `pkg_get_deps()` and `pkg_get_deps_minimal()`:
A character vector of direct dependencies, without duplicates.

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-deper

See Also

tinycodet_import

Examples

```
"dplyr" %installed in% .libPaths()

pkg_get_deps_minimal("dplyr")
pkgs <- pkg_get_deps("dplyr")
pkgs %installed in% .libPaths()
pkg_lsf("dplyr", "all")
```

pversion *Check for Package Versions Mismatch*

Description

The pversion_check4mismatch() function checks if there is any mismatch between the currently loaded packages and the packages in the specified library path.

The pversion_report() function gives a table of all specified packages, with their loaded and installed versions, regardless if there is a mismatch or not.

Usage

```
pversion_check4mismatch(pkgs = NULL, lib.loc = .libPaths())

pversion_report(pkgs = NULL, lib.loc = .libPaths())
```

Arguments

pkgs	a character vector with the package name(s). Packages that are not actually loaded will be ignored. Base/core R will also be ignored. If NULL, all loaded packages (see loadedNamespaces) excluding core/base R will be checked.
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

For pversion_check4mismatch():
If no mismatch between loaded versions and those in lib.loc were found, returns NULL.
Otherwise it returns a data.frame, with the loaded version and library version of the specified packages.

For pversion_report():
Returns a data.frame, with the loaded version and library version of the specified packages, as well as a logical column indicating whether the two versions are equal (TRUE), or not equal (FALSE).

See Also

tinycodet_import

Examples

```
"dplyr" %installed in%  .libPaths()

import_as(~dpr., "dplyr")
pversion_check4mismatch()
pversion_report()
```

`report_inops` *Report Infix Operators*

Description

The `report_inops()` function returns a data.frame listing the infix operators defined in the current environment, 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 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()
```

`safer_partialmatch` *Set Safer Dollar, Arguments, and Attribute Matching*

Description

The `safer_partialmatch()` function simply calls the following:

```
options(
   warnPartialMatchDollar = TRUE,
   warnPartialMatchArgs = TRUE,
   warnPartialMatchAttr = TRUE
 )
```

Thus it forces 'R' to give a warning when partial matching occurs when using the dollar ($) operator, or when other forms of partial matching occurs.
The `safer_partialmatch()` function is intended for when running R interactively (see interactive).

Usage

```
safer_partialmatch()
```

Value

Sets the options. Returns nothing.

See Also

tinycodet_safer

Examples

```
interactive()


safer_partialmatch()
data(iris)
head(iris)
iris$Sepal.Length <- iris$Sepal.Length^2
head(iris)
```

`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.

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`(..., tokens_only = FALSE)`) cuts every string into individual text breaks (like character, word, line, or sentence boundaries).

Usage

```
strcut_loc(str, loc)

strcut_brk(str, type = "character", tolist = FALSE, n = -1L, ...)
```

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`	either one of the following: • a single string giving the break iterator type (i.e. `"character"`, `"line_break"`, `"sentence"`, `"word"`, or a custom set of ICU break iteration rules). • a list with break iteration options, like a list produced by stri_opts_brkiter. about search: boundaries

tolist	logical, indicating if strcut_brk should return a list (TRUE), or a matrix (FALSE, default).
n	see stri_split_boundaries.
...	additional arguments to be passed to stri_split_boundaries.

Details

The strcut_ functions provide a short and concise way to cut strings into pieces, without removing the delimiters, which is an operation that lies at the core of virtually all boundaries-operations in 'stringi'.

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 return a matrix by default.

Value

For strcut_loc():
A character matrix with length(str) rows and 3 columns, where for every row i it holds the following:

- the first column contains the sub-string **before** loc[i,], or NA if loc[i,] contains NA;
- the second column contains the sub_string at loc[i,], or the uncut string if loc[i,] contains NA;
- the third and last column contains the sub-string **after** loc[i,], or NA if loc[i,] contains NA.

For strcut_brk(..., tolist = FALSE):
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.

For strcut_brk(..., tolist = TRUE):
A list with length(str) elements, where each element is a character vector containing the cut string.

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))
strcut_loc(x, c(5, NA))
strcut_loc(x, c(NA, 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)
strcut_brk(test, n = 1)
strcut_brk(test, "line", tolist = TRUE)
strcut_brk(test, "word", tolist = TRUE)
strcut_brk(test, "sentence", tolist = TRUE)

brk <- stringi::stri_opts_brkiter(
  type = "line"
)
strcut_brk(test, brk)
```

---

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 within 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 within 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)


#############################################################################
# sorting characters in strings ====

x <- c(paste(sample(letters), collapse = ""),
       paste(sample(letters), collapse = ""))
print(x)
mat <- strcut_brk(x)
rank <- stringi::stri_rank(as.vector(mat)) |>  matrix(ncol = ncol(mat))
sorted <- mat %row~% rank
sorted[is.na(sorted)] <- ""
print(sorted)
stri_join_mat(sorted, margin = 1)
stri_join_mat(sorted, margin = 2)


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

# sorting words ====
```

```
x <- c("2nd 3rd 1st", "Goodbye everyone")
print(x)
mat <- strcut_brk(x, "word")
rank <- stringi::stri_rank(as.vector(mat)) |> matrix(ncol = ncol(mat))
sorted <- mat %row~% rank
sorted[is.na(sorted)] <- ""
stri_c_mat(sorted, margin = 1, sep = " ") # <- alias for stri_join_mat
stri_c_mat(sorted, margin = 2, sep = " ")


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

# randomly shuffling 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 <- sample(seq_along(mat)) |> matrix(ncol = ncol(mat))
sorted <- mat %row~% rank
sorted[is.na(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_regex(str, pattern, i, ..., opts_regex = NULL)

stri_locate_ith_fixed(str, pattern, i, ..., opts_fixed = NULL)

stri_locate_ith_coll(str, pattern, i, ..., opts_collator = NULL)

stri_locate_ith_charclass(str, pattern, i, merge = TRUE, ...)

stri_locate_ith_boundaries(str, i, ..., opts_brkiter = NULL)
```

Arguments

`str`	a string or character vector.
`i`	an integer scalar, or an integer vector of appropriate length (vector recycling is not supported). Positive numbers count occurrences from the left/beginning of the strings. Negative numbers count occurrences from the right/end of the strings. 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 pattern occurrences `n`, the first (if `i < -n`) or last (if `i > n`) 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_all or stri_locate_all_boundaries. Do not supply the arguments `omit_no_match` or `get_length`, as they are already specified internally. Supplying these arguments anyway will result in an error.
`pattern`, `regex`, `fixed`, `coll`, `charclass`	a character vector of search patterns, as in stri_locate_all. about search: regex about search: fixed about search: coll about search: charclass
`opts_regex`, `opts_fixed`, `opts_collator`, `opts_brkiter`	named list used to tune up the selected search engine's settings. see stri_opts_regex, stri_opts_fixed, stri_opts_collator, and stri_opts_brkiter. NULL for the defaults. about search: regex about search: fixed about search: coll about search: charclass about search: boundaries
`merge`	logical, indicating if charclass locations should be merged or not. **Details:** For the `charclass` pattern type, the `stri_locate_ith()` function gives the start and end of **consecutive** characters by default, just like stri_locate_all.

To give the start and end positions of single characters, much like stri_locate_first or stri_locate_last, set `merge = FALSE`.

Details

The 'stringi' functions only support operations on the first, last, or all occurrences of a pattern.
The `stri_locate_ith()` function allows locating the i^{th} occurrence of a pattern.
This allows for several workflows for operating on the i^{th} pattern occurrence.
See also the examples section.

Extract i^{th} Occurrence of a Pattern

For extracting the i^{th} pattern occurrence:
Locate the the i^{th} occurrence using `stri_locate_ith()`, and then extract it using, for example, stri_sub.

Replace/Transform i^{th} Occurrence of a Pattern

For replacing/transforming the i^{th} pattern occurrence:

1. Locate the the i^{th} occurrence using `stri_locate_ith()`.
2. Extract the occurrence using stri_sub.
3. Transform or replace the extracted sub-strings.
4. Return the transformed/replaced sub-string back, using again stri_sub.

Capture Groups of i^{th} Occurrence of a Pattern

The `capture_groups` argument for `regex` is not supported within `stri_locate_ith()`.
To capture the groups of the i^{th} occurrences:

1. Use `stri_locate_ith()` to locate the i^{th} occurrences without group capture.
2. Extract the occurrence using stri_sub.
3. Get the matched group capture on the extracted occurrences using stri_match.

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`.

If an empty string or empty pattern is supplied, a warning is given and a matrix with 0 rows is returned.

Note

Long Vectors

The `stri_locate_ith`-functions do not support `long vectors` (i.e. character vectors with more

than 2^31 - 1 strings).

Performance
The performance of stri_locate_ith() is about the same as that of stri_locate_all.

See Also

tinycodet_strings

Examples

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

# practical example: transform regex pattern ====

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

# locate 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, loc)
print(extr)

# transform & replace ith vowels with numbers:
repl <- chartr("aeiou", "12345", extr)
stringi::stri_sub(x, loc) <- repl

# result (notice ith vowels are now numbers):
print(x)

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


# practical example: group-capture regex pattern ====

# input character:
# first group: c(breakfast=eggs, breakfast=bacon)
# second group: c(lunch=pizza, lunch=spaghetti)
x <- c('breakfast=eggs;lunch=pizza',
       'breakfast=bacon;lunch=spaghetti',
       'no food here') # no group here
print(x)

# locate ith=2nd group:
```

```
p <- '(\\w+)=(\\w+)'
loc <- stri_locate_ith(x, i = 2, regex = p)
print(loc)

# extract ith=2nd group:
extr <- stringi::stri_sub(x, loc)
print(extr)

# capture ith=2nd group:
stringi::stri_match(extr, regex = p)

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


# practical example: replace words using 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 (notice ith words have inverted case):
tf <- chartr(extr, old = "a-zA-Z", new = "A-Za-z")
stringi::stri_sub(x, loc) <- tf

# result:
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 character/pattern defined in `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.

The `x %ss% p` operator splits the strings in `x` by a delimiter character/pattern defined in `p`, and removes `p` in the process.
For cutting strings by text boundaries, or around a location, see strcut_brk and strcut_loc.

Usage

```
x %s-% p

x %s/% p

x %s//% brk

x %ss% p
```

Arguments

x	a string or character vector.
p	either a list with 'stringi' arguments (see s_pattern), or else a character vector with regular expressions. about search: regex about search: fixed about search: coll about search: charclass
brk	a list with break iteration options, like a list produced by 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.
The %ss% operator returns a list of the split strings - or, if simplify = TRUE / simplify = NA, returns a matrix of the split strings.

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
x %ss% p # split x around vowels, removing the vowels in the process
x %ss% s_regex(p, simplify = NA) # same as above, but in matrix form

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.

x <- c(paste(letters, collapse = ", "), paste(LETTERS, collapse = ", "))
print(x)
x %ss% ", "
t(x %ss% s_fixed(", ", simplify = NA))
```

`str_search` *'stringi' Pattern Search Operators*

Description

The `x %s{}% p` **and** `x %s!{}% p` **Operators:**
The `x %s{}% p` operator checks for every string in character vector `x` if the pattern defined in `p` is present.
When supplying a list on the right hand side (see s_pattern), one can optionally include the list element `at = "start"` or `at = "end"`:

- Supplying `at = "start"` will check if the pattern appears at the start of a string (like stri_startswith).
- Supplying `at = "end"` will check if the pattern appears at the end of a string (like stri_endswith).

The `x %s!{}% p` operator is the same as `x %s{}% p`, except it checks for **absence** of the pattern, rather than presence.

For string (in)equality operators, see %s==% from the 'stringi' package.

`strfind()<-`**:**
`strfind()<-` locates, extracts, or replaces found patterns.
It complements the other string-related operators, and uses the same s_pattern API.
It functions as follows:

- `strfind()` finds all pattern matches, and returns the extractions of the findings in a list, just like stri_extract_all.
- `strfind(..., i = "all")`, finds all pattern matches like stri_locate_all.
- `strfind(..., i = i)`, where `i` is an integer vector, locates the i^{th} occurrence of a pattern, and reports the locations in a matrix, just like stri_locate_ith.

- `strfind() <- value` finds pattern matches in variable `x`, replaces the pattern matches with the character vector specified in `value`, and assigns the transformed character vector back to `x`.
  This is somewhat similar to stri_replace, though the replacement is done in-place.

Usage

```
x %s{}% p

x %s!{}% p

strfind(x, p, ..., i, rt)

strfind(x, p, ..., i, rt) <- value
```

Arguments

`x`	a string or character vector. For `strfind()<-`, `x` must obviously be the variable containing the character vector/string, since `strfind()<-` performs assignment in-place.
`p`	either a list with 'stringi' arguments (see s_pattern), or else a character vector with regular expressions. See also the Details section. about search: regex about search: fixed about search: coll about search: charclass
`...`	additional arguments to be specified.
`i`	either one of the following can be given for `i`: • if `i` is not given or `NULL`, `strfind()` extracts all found pattern occurrences. • if `i` is the string "all", `strfind()` locates all found pattern occurrences. • if `i` is an integer, `strfind()` locates the i^{th} pattern occurrences. See the `i` argument in stri_locate_ith for details. For `strfind() <- value`, `i` must not be specified.
`rt`	use `rt` to specify the Replacement Type that `strfind()<-` should perform. Either one of the following can be given for `rt`: • if `rt` is not given, `NULL` or `"vec"`, `strfind()<-` performs regular, vectorized replacement of **all** occurrences. • if `rt` = `"dict"`, `strfind()<-` performs dictionary replacement of **all** occurrences. • if `rt` = `"first"`, `strfind()<-` replaces only the first occurrences.

- if `rt = "last"`, `strfind()<-` replaces only the last occurrences.

Note: `rt = "first"` and `rt = "last"` only exist for convenience; for more specific locational replacement, use stri_locate_ith or `strfind(..., i)` with numeric `i` (see the Examples section).
For `strfind()`, `rt` must not be specified.

`value` a character vector giving the replacement values.

Details

Right-hand Side List for the `%s{}%` and `%s!{}%` Operators

When supplying a list to the right-hand side of the `%s{}%` and `%s!{}%` operators, one can add the argument `at`.
If `at = "start"`, the operators will check if the pattern is present/absent at the start of the string.
If `at = "end"`, the operators will check if the pattern is present/absent at the end of the string.
Unlike stri_startswith or stri_endswith, `regex` **is** supported by the `%s{}%` and `%s!{}%` operators.
See examples below.

Vectorized Replacement vs Dictionary Replacement

- Vectorized replacement:
  `x`, `p`, and `value` are of the same length (or recycled to become the same length).
  All occurrences of pattern `p[j]` in `x[j]` is replaced with `value[j]`, for every `j`.
- Dictionary replacement:
  `p` and `value` are of the same length, and their length is independent of the length of `x`.
  For every single string in `x`, all occurrences of pattern `p[1]` are replaced with `value[1]`, all occurrences of pattern `p[2]` are replaced with `value[2]`, etc.

Notice that for single replacement, i.e. `rt = "first"` or `rt = "last"`, it makes no sense to distinguish between vectorized or dictionary replacement, since then only a single occurrence is being replaced per string.
See examples below.

Value

For the `x %s{}% p` and `x %s!{}% p` operators:
Return logical vectors.

For `strfind()`:
Returns a list with extractions of all found patterns.

For `strfind(..., i = "all")`:
Returns a list with all found pattern locations.

For `strfind(..., i = i)` with integer vector `i`:
Returns an integer matrix with two columns, giving the start and end positions of the i^{th} matches, two NAs if no matches are found, and also two `NA`s if str is `NA`.

For `strfind() <- value`:
Returns nothing, but performs in-place replacement (using R's default in-place semantics) of the found patterns in variable `x`.

Note

`strfind()<-` performs in-place replacement.
Therefore, the character vector or string to perform replacement on, must already exist as a variable.
So take for example the following code:

```
strfind("hello", p = "e") <- "a" # this obviously does not work

y <- "hello"
strfind(y, p = "e") <- "a" # this works fine
```

In the above code, the first `strfind()<-` call does not work, because the string needs to exist as a variable.

See Also

tinycodet_strings

Examples

```
# example of %s{}% and %s!{}% ====

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)

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


# Example of %s{}% and %s!{}% with "at" argument ====

x <- c(paste0(letters, collapse = ""),
       paste0(rev(letters), collapse = ""), NA)
p <- s_fixed("abc", at = "start")
x %s{}% p
stringi::stri_startswith(x, fixed = "abc") # same as above

p <- s_fixed("xyz", at = "end")
x %s{}% p
stringi::stri_endswith(x, fixed = "xyz") # same as above

p <- s_fixed("cba", at = "end")
x %s{}% p
stringi::stri_endswith(x, fixed = "cba") # same as above

p <- s_fixed("zyx", at = "start")
x %s{}% p
stringi::stri_startswith(x, fixed = "zyx") # same as above



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


# Example of transforming ith occurrence ====

# new 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 <- s_regex( # vowels
  rep("A|E|I|O|U", 2),
  case_insensitive = TRUE
)
loc <- strfind(x, p, i = c(2, -2))
print(loc)

# extract ith vowels:
```

```
extr <- stringi::stri_sub(x, from = loc)
print(extr)

# replace ith vowels with numbers:
repl <- chartr("aeiou", "12345", extr) # transformation
stringi::stri_sub(x, loc) <- repl
print(x)


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


# Example of strfind for regular vectorized replacement ====

x <- rep('The quick brown fox jumped over the lazy dog.', 3)
print(x)
p <- c('quick', 'brown', 'fox')
rp <- c('SLOW',  'BLACK', 'BEAR')
x %s{}% p
strfind(x, p)
strfind(x, p) <- rp
print(x)

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


# Example of strfind for dictionary replacement ====

x <- rep('The quick brown fox jumped over the lazy dog.', 3)
print(x)
p <- c('quick', 'brown', 'fox')
rp <- c('SLOW',  'BLACK', 'BEAR')
# thus dictionary is:
# quick => SLOW; brown => BLACK; fox => BEAR
strfind(x, p, rt = "dict") <- rp
print(x)


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


# Example of strfind for first and last replacement ====

x <- rep('The quick brown fox jumped over the lazy dog.', 3)
print(x)
p <- s_fixed("the", case_insensitive = TRUE)
rp <- "One"
strfind(x, p, rt = "first") <- rp
print(x)

x <- rep('The quick brown fox jumped over the lazy dog.', 3)
print(x)
p <- s_fixed("the", case_insensitive = TRUE)
```

```
rp <- "Some Other"
strfind(x, p, rt = "last") <- rp
print(x)
```

`str_subset_ops` *String Subsetting Operators*

Description

String subsetting operators.

The `x %s><% ss` operator gets a certain number of the first and last characters of every string in character vector `x`.
`%sget%` is an alias for `%s><%`.

The `x %s<>% ss` operator trims a certain number of the first and last characters of every string in character vector `x`.
`%strim%` is an alias for `%<>%`.

Usage

```
x %s><% ss

x %s<>% ss

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

Both operators return a character vector of the same length as `x`.

The `x %s><% ss` operator gives a certain number of the first and last characters of each string in the input character vector `x`.

The `x %s<>% ss` operator removes a certain number of the first and last characters of each string in the input 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 %s><% ss

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

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

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

s_pattern *Pattern Specifications for String Related Operators*

Description

The %s-%, %s/%, %ss% operators, as well as the string search operators (str_search), perform pattern matching for some purpose, where the pattern is given in the second argument (p).
When a character vector or string is given as the second argument (p), 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 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: the `at` argument for the str_search operators.
`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 operators.

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

x <- c(paste0(letters[1:13], collapse = ""),
      paste0(letters[14:26], collapse = ""), NA)
p <- s_fixed("abc", at = "start")
x %s{}% p
stringi::stri_startswith(x, fixed = "abc") # same as above

p <- s_fixed("xyz", at = "end")
x %s{}% p
stringi::stri_endswith(x, fixed = "xyz") # same as above
```

transform_if *transform_if: Conditional Sub-set Transformation of Atomic objects*

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.

For a more ifelse-like function where yes, no, and other are vectors, see kit::iif.

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 yes is 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
```

---

with_pro *Standard Evaluated Versions of Some Common Expression-Evaluation Functions*

Description

The `with_pro()` and `aes_pro()` functions are standard-evaluated versions of the expression-evaluation functions with and `ggplot2::aes`, respectively.

These alternative functions are more programmatically friendly:
They use proper standard evaluation, through the usage of one-sided formulas, instead of non-standard evaluation, tidy evaluation, or similar programmatically unfriendly evaluations.

Usage

```
with_pro(data, form)

aes_pro(...)
```

Arguments

data	a list, environment, or data.frame.
form	a one-sided formula giving the expression to evaluate in with_pro. If the formula has an environment, that environment is used to find any variables or objects not present in data.
...	arguments to be passed to ggplot2::aes, but given as one-sided formulas.

Details

The aes_pro() function is the standard evaluated alternative to ggplot2::aes.
Due to the way aes_pro() is programmed, it should work even if the tidy evaluation technique changes in 'ggplot2'.
To support functions in combinations with references of the variables, the input used here are formula inputs, rather than string inputs.
See the Examples section below.

Value

For with_pro(): see with.
For aes_pro(): see ggplot2::aes.

Non-Standard Evaluation

Non-Standard Evaluation (sometimes abbreviated as "NSE"), is somewhat controversial.
Consider the following example:

```
aplot <- "ggplot2"
library(aplot)
```

What package will be attached? It will not be 'ggplot2', nor will an error occur. Instead, the package 'aplot' will be attached.
This is due to evaluating the expression 'aplot' as a quoted expression, instead of evaluating the contents (i.e. string or formula) of the variable. In other words: Non-Standard Evaluation.

Regular Standard Evaluation does not have the above problem.

Note

The `with_pro()` function, like the original with function, is made for primarily for convenience. When using modelling or graphics functions with an explicit `data` argument (and typically using formulas), it is typically preferred to use the `data` argument of that function, rather than to use either `with(data, ...)` or `with_pro(data, ...)`.

See Also

tinycodet_safer

Examples

```
requireNamespace("ggplot2")


d <- import_data("ggplot2", "mpg")

# mutate data:
myform <- ~ displ + cyl + cty + hwy
d$mysum <- with_pro(d, myform)
summary(d)

# plotting data:
x <- ~ cty
y <- ~ sqrt(hwy)
color <- ~ drv

ggplot2::ggplot(d, aes_pro(x, y, color = color)) +
  ggplot2::geom_point()
```

---

xxx_atomic_typecast *DEPRECATED: Atomic Type Casting With Names and Dimensions Preserved*

Description

THESE FUNCTIONS ARE DEPRECATED, AND WILL BE REMOVED IN A FUTURE VERSION.
Please use the functions of the same name from the 'broadcast' package instead.

Usage

```
as_bool(x, ...)

as_int(x, ...)

as_dbl(x, ...)

as_chr(x, ...)

as_cplx(x, ...)

as_raw(x, ...)
```

Arguments

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

Value

The converted object.

Examples

```
# matrix example ====
x <- matrix(sample(-1:28), ncol = 5)
colnames(x) <- month.name[1:5]
rownames(x) <- month.abb[1:6]
names(x) <- c(letters[1:20], LETTERS[1:10])
print(x)

as_bool(x)
as_int(x)
as_dbl(x)
as_chr(x)
as_cplx(x)
as_raw(x)


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

# factor example ====
x <- factor(month.abb, levels = month.abb)
names(x) <- month.name
print(x)

as_bool(as_int(x) > 6)
as_int(x)
```

```
as_dbl(x)
as_chr(x)
as_cplx(x)
as_raw(x)
```

Index