Package 'subsets'

October 31, 2023

Type Package

Title Grammar of Subsets

Version 0.0.0.9

Date 2023-10-13

Description Subsetting methods for various data-types.
These methods have the following properties.
1) Programmatically friendly.
2) Beginner friendly.
3) Class consistent.
4) Careful handling of name-based indexing.
5) Performance aware.
6) Support vector-like atomic objects (vectors, matrices, arrays).
7) Support factors.
8) Support lists.
9) Support data.frame-like objects (data.frame, data.table, tibble, tidytable, etc.).
10) Support ggplot2 aesthetics.

License GPL (>= 2)

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests rlang,
knitr,
rmarkdown,
tinytest,
abind,
tinycodet,
tidytable,
tibble,
dplyr,
ggplot2

Depends R (>= 4.1.0)

Imports methods,
Rcpp (>= 1.0.11),
collapse (>= 2.0.2),
data.table (>= 1.14.8)

Encoding UTF-8

Language en-GB
VignetteBuilder knitr

R topics documented:

`aaa0_subsets` *subsets: an Easy Grammar of Subsets*

Description

subsets: an Easy Grammar of Subsets

Motivation

Among programming languages, 'R' has perhaps one of the most flexible and comprehensive sub-setting functionality. But with flexibility often comes confusion and (apparent) inconsistencies. And R is no exception.

This becomes quite apparent when one reads (online) documents such as "The R Inferno" by Patrick Burns, and "Frustration: One Year With R" by Reece Goding. These document point out many inconsistencies, and sub-setting related inconsistencies make up a good portion of these documents.

To my surprise, there is no comprehensive R package (as far as I could see at least) that actually attempts to "fix" the subset-related issues laid out in these and other such documents.

Famous subset-related R packages such as 'dplyr' and 'data.table' focus almost exclusive on data.frame-like objects, and occasionally even add more frustration in some aspects, like being not very programmatically friendly.

Thus, this R package was born.

Although this package was somewhat made for people who are new to 'R' (especially when coming from another programming language), and found themselves confused, I trust this package will be useful even for those who are quite experienced in 'R'.

Properties

The subsets package provides easier subsetting functions with the following properties:

- **Programmatically friendly**:
  - Non-standard evaluation is highly controversial (and for good reasons!), and therefore completely absent in this R package;
  - Name-based instead of position-based arguments;
  - Missing arguments can be filled with `NULL`, instead of using dark magic like `base::quote(expr = )`.
  - Functions are pipe-friendly.
- **Beginner friendly**:
  - No (silent) vector recycling;
  - Extracting and removing subsets uses the same syntax.
- **Class consistent**:
  - sub-setting of multi-dimensional objects by specifying dimensions (i.e. rows, columns, ...) use `drop = FALSE`. So matrix in, matrix out.
  - The functions deliver the same results for data.frames, data.tables, tibbles, and tidytables. No longer does one have to re-learn the different brackets-based sub-setting rules for different types of data.frame-like objects. Powered by the subclass agnostic 'C'-code from 'collapse' and 'data.table'.
  - Smart with sub-setting recursive lists.
- **Careful handling of name-based indexing**:
  - Sub-setting object by index names returns ALL indices with that name, not just the first;
  - Data.frame-like objects are forced to have unique column names;
  - Selecting non-existing names always gives an error.
- **Support a wide variety of data types**:
  - Support vector-like (atomic) objects (vectors, matrices, arrays);
  - Support lists;
  - Support factors;
  - Support data.frame-like objects (data.frame, data.table, tibble, tidytable, etc.).
- **Concise function and argument names**.
- **Special functions**:
  for string subsetting, vectorized recursive list subsetting, and even for the column selection subsetting used in ggplot2's aes function.
- **Performance aware**:
  Despite the many checks performed, the functions are kept reasonably speedy, through the use of the 'Rcpp', 'collapse', and 'data.table' R-packages.

Methods and Functions

The main focus is on the following generic S3 methods:

- sb_x: method to extract, exchange, or duplicate indices.
- sb_rm: method to remove indices.
- sb_tf: method to transform values of subsets.

- sb_rp: method to replace values of subsets.
- sb_before, sb_after: methods to insert new values before or after an index along a dimension of an object.

Beside these generic S3 methods, additional specialized sub-setting functions are provided:

- aes_pro: programmatically friendly and stable version of ggplot2's aesthetic sub-setting function.
- sb_str: subset a single string (each single character is treated as a single element).
- sb_rec: recursive sub-setting of lists.

And finally, a couple of helper functions for creating ranges and sequences (occasionally needed in sub-setting) are provided:

- seq_rec: Generalized recursive sequence generator.
- seq_names: create a range of indices from a specified starting and ending name.

Author(s)

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

See Also

'subsets' relies on the 'Rcpp', 'collapse' and 'data.table' R-packages to ensure an acceptable performance of its functions despite the many checks that these functions perform. I also recommend using these packages for other subsetting and data wrangling functionalities.
Besides these package, the following R packages work very nicely together with 'subsets':

- 'stringi':
  THE R package for fast and concise string manipulation - an essential part of any programming language.
- 'abind':
  Provides binding arrays along an arbitrary dimension.
- 'tinycodet':
  Helps the user with their coding etiquette. Focuses on 4 aspects: (1) safe functionalities, (2) an import system that combines benefits of using without attaching and attaching a package, (3) extending the capabilities of the aforementioned 'stringi' package, (4) functions to reduce repetitive code.

---

aes_pro *Programmatically Friendly, Standard Evaluated aes() Function Alias*

Description

Programmatically friendly version of 'ggplot2"s aes function. This function is programmatically friendly because it uses proper standard evaluation, instead of non-standard evaluation, tidy evaluation, or similar programmatically unfriendly evaluations.

Usage

```
aes_pro(...)
```

Arguments

...	arguments to be passed to aes, except `aes_pro()` forces programmatically friendly standard evaluation.

Details

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

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

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

Often standard-evaluated alternatives are also provided. But in the case of the `aes()` function in 'ggplot2', the standard-evaluated alternative changes frequently. Moreover, the alternatives provided so far are rather clumsy.

The `aes_pro()` function is the standard evaluated alternative. Due to the way `aes_pro()` is programmed, it should work no matter how many times the standard evaluation techniques change in 'ggplot2'. It should also work in older and newer versions of 'ggplot2'.

Value

See aes.

Examples

```
requireNamespace("ggplot2")


data("starwars", package = "dplyr")
x <- "mass"
y <- "height"
color <- "sex"
xform <- ~ mass

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

indx_x *Exported Utilities*

Description

Exported utilities

Usage

```
indx_x(i, x, xnames, xsize)

indx_rm(i, x, xnames, xsize)
```

Arguments

i	any of the following: • a vector of length 0, in which case an empty object is returned. • a strictly positive vector with indices (duplicates are allowed, resulting in duplicated indices). • logical vector (without NAs) of the same length as x giving the indices to select. • a character vector of index names (duplicates are allowed, resulting in duplicated indices). If an object has multiple indices with the given name, all the corresponding indices will be selected/duplicated/re-arranged. • function that returns a logical vector giving the element indices to select.
x	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.
xnames	names or dimension names
xsize	length or dimension size

Value

The subsetted object.

Examples

```
x <- 1:10
names(x) <- letters[1:10]
indx_x(1:5, x, names(x), length(x))
indx_rm(1:5, x, names(x), length(x))
```

rcpp_hello_world *Simple function using Rcpp*

Description

Simple function using Rcpp

Usage

```
rcpp_hello_world()
```

Examples

```
## Not run:
rcpp_hello_world()

## End(Not run)
```

sb_before *Methods to insert new values before or after an index along a dimension*

Description

The sb_before() method inserts new values before some position along a dimension.
The sb_after() method inserts new values after some position along a dimension.

Usage

```
sb_before(x, ...)

sb_after(x, ...)

## Default S3 method:
sb_before(x, new, pos = 1, .attr = attributes(x), ...)

## Default S3 method:
sb_after(x, new, pos = length(x), .attr = attributes(x), ...)

## S3 method for class 'factor'
sb_before(x, new, pos = 1, .attr = attributes(x), ...)

## S3 method for class 'factor'
sb_after(x, new, pos = length(x), .attr = attributes(x), ...)

## S3 method for class 'list'
sb_before(x, new, pos = 1, .attr = attributes(x), ...)

## S3 method for class 'list'
```

```
sb_after(x, new, pos = length(x), .attr = attributes(x), ...)

## S3 method for class 'array'
sb_before(x, new, margin, pos = 1, .attr = attributes(x), ...)

## S3 method for class 'array'
sb_after(x, new, margin, pos = dim(x)[margin], .attr = attributes(x), ...)

## S3 method for class 'data.frame'
sb_before(x, new, margin, pos = 1, .attr = attributes(x), ...)

## S3 method for class 'data.frame'
sb_after(
  x,
  new,
  margin,
  pos = collapse::fdim(x)[margin],
  .attr = attributes(x),
  ...
)
```

Arguments

x	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.
...	further arguments passed to or from other methods.
new	the new value(s). The type of object depends on x: • For vector-like objects, new can be any vector-like object. However, if one wished the added values in new to be named, it is advised to ensure new is the same type of object as x. (I.e. use matrix with column names for new when appending/inserting columns to matrix x.) • For factors, new must be a factor. • For lists, new must be a (possible named) list. • For data.frame-like objects, new must be a data.frame.
pos	a strictly positive single integer scalar (so no duplicates), giving the position along the dimension (specified in margin), before or after which the new values are added.
.attr	a list, giving additional potentially missing attributes to be added to the returned object. By default, R's c function concatenates AND strips attributes (for good reasons). In the attr argument, the attributes of the merged object can be specified. Only attributes that are actually missing AFTER insertion will be added, thus preventing overwriting existing attributes like names. Defaults to the attributes of x. If NULL, no attributes will be added post-insert. If speed is important, NULL is the best option (but then attributes won't be preserved).
margin	a single scalar, giving the dimension along which to add new values.

Value

Returns a copy of the appended object.

Examples

```
# vector-like objects ====
x <- matrix(1:20 , ncol = 4)
print(x)
new <- -1 * x
sb_before(x, new, 1)
sb_before(x, new, 2)
sb_after(x, new, 1)
sb_after(x, new, 2)


# lists ====
x <- as.list(1:5)
new <- lapply(x, \(x)x*-1)
print(x)
sb_before(x, new)
sb_after(x, new)


# factors ====
x <- factor(letters)
new <- factor("foo")
sb_before(x, new)
sb_after(x, new)


# data.frame-like objects ====
x <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
new <- data.frame(e = 101:110)
sb_before(x, new, 2)
sb_after(x, new, 2)
new <- x[1,]
sb_before(x, new, 1)
sb_after(x, new, 1)
```

`sb_rm` *Methods to remove Subsets from an object*

Description

This is an S3 Method to remove subsets from an object.

Usage

```
sb_rm(x, ...)

## Default S3 method:
sb_rm(x, i, ...)

## S3 method for class 'factor'
```

```
sb_rm(x, i = NULL, lvl = NULL, drop = FALSE, ...)

## S3 method for class 'list'
sb_rm(x, i, drop = FALSE, ...)

## S3 method for class 'matrix'
sb_rm(x, row = NULL, col = NULL, i = NULL, ...)

## S3 method for class 'array'
sb_rm(x, idx = NULL, dims = NULL, i = NULL, ...)

## S3 method for class 'data.frame'
sb_rm(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ...)
```

Arguments

`x`	vector, matrix, array, data.frame, or list
`...`	further arguments passed to or from other methods.
`i`	`sb_rm(x, i = i)` corresponds to `x[i]` - except `sb_rm()` removes indices. Any of the following can be given here: • `NULL`, only for multi-dimensional objects or factors. `NULL` results in nothing being removed, and the entire object is returned. • a vector of length 0, in which case nothing is removed, and the entire object is returned. • a strictly positive integer vector with indices to remove (duplicates are NOT allowed). • logical vector (without `NA`s) of the same length as `x` giving the indices to remove. • a character vector of index names to remove (duplicates are NOT allowed). If an object has multiple indices with the given name(s), all the corresponding indices will be removed. • a function that returns a logical vector giving the element indices to remove.
`lvl`	names of the levels to remove. Duplicates are NOT allowed.
`drop`	logical. • For factors: If `TRUE`, unused levels are dropped, if `FALSE` they are not dropped. • For lists: is `TRUE`, selecting a single element will give the simplified result, like using `[[]]`. If `FALSE`, a list is always returned regardless of the number of elements.
`row, col`	`sb(x, row, col)` corresponds to `x[row, col, drop = FALSE]` - except `sb_rm()` removes indices. Thus `row` = rows, `col` = columns, `lyr` = layers (i.e. third dimension). Any of the following can be given here: • `NULL` (default), which results in ALL indices of this dimension being returned. • a vector of length 0, in which case nothing is removed and the entire object is returned. • a strictly positive integer vector with indices (duplicates are NOT allowed).

- logical vector (without NAs) of the same length as the corresponding dimension size, giving the indices of this dimension to remove.
- a character vector of index names (duplicates are NOT allowed). If an object has multiple indices with the given name(s), all the corresponding indices will be removed.

NOTE: The arguments row and col will be ignored if i is specified.

idx, dims — arguments to subset arrays:

- idx: a list of indices.
- dims: a integer vector of the same length as idx, giving the dimensions to which the indices given in idx correspond to.

The elements of idx follow the same rules as the rules for row and col, EXCEPT one should not fill in NULL.
Thus sb_rm(x, list(1:10, 1:4), c(1, 3)) is equivalent to x[1:10, , 1:4, drop = FALSE], except sb_rm() removes indices.
NOTE: The arguments idx and dims will be ignored if i is specified.

filter — a one-sided formula with a single logical expression using the column names of the data.frame, giving the condition which observations (rows) should be removed.
For example:
to **remove** rows for which column a > 2 and for which column b != "a", specify the following formula:
~ (a > 2) & (b != "a")

vars — a function, giving the condition which variables (columns) should be removed.

Details

One cannot specify i and row/col/lvl/idx/dims simultaneously. It's either i, or the other arguments.

One cannot specify row and filter simultaneously. It's either one or the other. Similarly, one cannot specify col and vars simultaneously.

In the above cases it holds that if one set is specified, the other is set is ignored.

Value

A copy of the sub-setted object.

Examples

```
# vector-like objects ====
obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_rm(obj, 1:3, 1:3)
# above is equivalent to  obj[-1:-3, -1:-3, drop = FALSE]
sb_rm(obj, i = \(x)x>5)
# above is equivalent to  obj[!obj > 5]
sb_rm(obj, col = "a")
# above is equivalent to  obj[, which(!colnames(obj) %in% "a")]

obj <- array(1:64, c(4,4,3))
```

```
print(obj)
sb_rm(obj, list(1, 1, c(1, 3)), 1:3)
# above is equivalent to obj[-1, -1, c(-1, -3), drop = FALSE]
sb_rm(obj, i = \(x)x>5)
# above is equivalent to obj[!obj > 5]


# lists ====
obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_rm(obj, "a")
# above is equivalent to obj[which(!names(obj) %in% "a")]
sb_rm(obj, 1) # obj[-1]
sb_rm(obj, 1:2)
# above is equivalent to obj[[seq_len(length(obj))[-1:-2]]]
sb_rm(obj, is.numeric, drop = TRUE)
# above is equivalent to obj[[!sapply(obj, is.numeric)]] IF this returns a single element
obj <- list(a = 1:10, b = letters[1:11], c = letters)
sb_rm(obj, is.numeric)
# above is equivalent to obj[!sapply(obj, is.numeric)] # this time singular brackets?
# for recusive indexing, see sb_rec()


# factors ====
obj <- factor(rep(letters[1:5], 2))
sb_rm(obj, lvl = "a")
# above is equivalent to obj[which(!obj %in% "a")]


# data.frame-like objects ====
obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb_rm(obj, 1:3, 1:3)
# above is equivalent to obj[-1:-3, -1:-3, drop = FALSE]
sb_rm(obj, filter = ~ (a > 5) & (c < 19), vars = is.numeric)
```

---

sb_rp *Method to Replace Subsets of an Object With Different Values*

Description

This is an S3 Method to replace a subset of an object with different values.

Usage

```
sb_rp(x, ...)

## Default S3 method:
sb_rp(x, i, ..., rp)

## S3 method for class 'list'
```

```
sb_rp(x, i, ..., rp)

## S3 method for class 'factor'
sb_rp(x, i = NULL, lvl = NULL, ..., rp)

## S3 method for class 'matrix'
sb_rp(x, row = NULL, col = NULL, i = NULL, ..., rp)

## S3 method for class 'array'
sb_rp(x, idx = NULL, dims = NULL, i = NULL, ..., rp)

## S3 method for class 'data.frame'
sb_rp(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ..., rp)
```

Arguments

`x`	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.
`...`	further arguments passed to or from other methods.
`i`	`sb_rp(x, i = i, rp = rp)` corresponds to `x[i] <- rp`. Any of the following can be given here: • `NULL`, only for multi-dimensional objects or factors. `NULL` results in the entire object being transformed. • a vector of length 0, in which case the original object is returned unchanged. • a strictly positive numeric vector with indices (duplicates are NOT allowed). • logical vector (without `NA`s) of the same length as `x` giving the indices to transform. • a character vector of index names (duplicates are NOT allowed). If an object has multiple indices with the given name, all the values of the corresponding indices will be transformed. • function that returns a logical vector giving the element indices to select.
`rp`	an object of somewhat the same type as the selected subset of `x`, and the same same length as the selected subset of `x` or a length of 1.
`lvl`	the levels of factor `x` to replace with the replacement vector `rp`. Note that if argument `i` is specified instead of `lvl`, re-levelling does not occur, and replacing subsets with new (i.e. unknown) levels will result in NA values.
`row, col`	`sb(x, row, col, rp = rp)` corresponds to `x[row, col] <- rp`. Thus `row` = rows, `col` = columns. Any of the following can be given here: • `NULL` (default), which results in ALL of the indices this dimension being transformed. • a vector of length 0, in which case the original object is returned unchanged. • a strictly positive integer vector with dimension indices to transform (duplicates are NOT allowed). • logical vector (without `NA`s) of the same length as the corresponding dimension size, giving the indices of this dimension to transform. • a character vector of index names (duplicates are NOT allowed). If an object has multiple indices with the given name, all the corresponding indices will be transformed. NOTE: The arguments `row` and `col` will be ignored if `i` is specified.

idx, dims — arguments to subset arrays:

- idx: a list of indices.
- dims: a integer vector of the same length as idx, giving the dimensions to which the indices given in idx correspond to.

The elements of idx follow the same rules as the rules for row and col, EXCEPT one should not fill in NULL.
Thus sb_rm(x, list(1:10, 1:4), c(1, 3), rp = rp) is equivalent to x[1:10, , 1:4] <- rp.
NOTE: The arguments idx and dims will be ignored if i is specified.

filter — a one-sided formula with a single logical expression using the column names of the data.frame, giving the condition which observations (rows) should be transformed.

vars — a function, giving the condition which variables (columns) should be transformed.

Details

One cannot specify i and row/col/lvl/idx/dims simultaneously. It's either i, or the other arguments.

One cannot specify row and filter simultaneously. It's either one or the other. Similarly, one cannot specify col and vars simultaneously.

In the above cases it holds that if one set is specified, the other is set is ignored.

Value

A copy of the object with replaced values.

Examples

```
# vector-like objects ====
obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_rp(obj, 1:3, 1:3, rp = -1:-9)
# above is equivalent to  obj[1:3, 1:3] <- -1:-9; obj
sb_rp(obj, i = \(x)x<=5, rp = -1:-5)
# above is equivalent to  obj[obj <= 5] <- -1:-5; obj
sb_rp(obj, col = "a", rp = -1:-8)
# above is equivalent to  obj[, which(colnames(obj) %in% "a")] <- -1:-8; obj

obj <- array(1:64, c(4,4,3))
print(obj)
sb_rp(obj, list(1:3, 1:2, c(1, 3)), 1:3, rp = -1:-12)
# above is equivalent to obj[1:3, 1:2, c(1, 3)] <- -1:-12
sb_rp(obj, i = \(x)x<=5, rp = -1:-5)
# above is equivalent to obj[obj <= 5] <- -1:-5


# lists ====
obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_rp(obj, "a", rp = list(1))
```

```
# above is equivalent to  obj[["a"]] <- 1; obj
sb_rp(obj, is.numeric, rp = list(-1:-10, -11:-20))
# above is equivalent to  obj[which(sapply(obj, is.numeric))] <- list(-1:-10, -11:-20); obj


# data.frame-like objects ====
obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb_rp(obj, vars = is.numeric, rp = data.frame(-1:-10))
```

`sb_special` *Specialized subsetting functions*

Description

The `sb_rec()` function performs recursive subsetting of lists.
The `sb_str()` function subsets a single string as-if it is an iterable object.

Usage

```
sb_rec(lst, rec)

sb_str(str, ind)
```

Arguments

`lst`	a list.
`rec`	a vector of length p, such that `lst[[rec]]` is equivalent to `lst[[rec[1]]...[[rec[p]]]`, providing all but the final indexing results in a list.
`str`	a single string.
`ind`	an integer vector, giving the positions of the string to subset.

Details

The `sb_str()` function is several times faster (the exact ratio depends on the string length) than using something like the following:

```
x <- strsplit(x, "") |> unlist()
x <- paste0(x[ind])
paste(x, collapse = "")
```

Value

The subsetted object.

Examples

```
lst <- list(
  A = list(
    A = list(A = "AAA", B = "AAB"),
    B = list(A = "ABA", B = "ABB")
  ),
  B = list(
    A = list(A = "BAA", B = "BAB"),
    B = list(A = "BBA", B = "BBB")
  )
)
sb_rec(lst, c(1,2,2)) # this gives "ABB"
sb_rec(lst, c(2,2,1)) # this gives "BBA"

x <- "hello"
sb_str(x, 5:1) # this gives "olleh"
sb_str(x, c(1:5, 5)) # this gives "helloo"
sb_str(x, c(2:5)) # this gives "ello"
sb_str(x, seq(1, 5, by = 2)) # this gives "hlo"
substring(x, 1:5, 3:7) # "hel" "ell" "llo" "lo"  "o"
```

---

sb_tf *Method to Transform a Subset of an Object*

Description

This is an S3 Method to transform a subset of an object.
Note that there is no `sb_tf` method for factors. This is intentional; use relevel instead.

Usage

```
sb_tf(x, ...)

## Default S3 method:
sb_tf(x, i, ..., tf)

## S3 method for class 'list'
sb_tf(x, i, ..., tf)

## S3 method for class 'matrix'
sb_tf(x, row = NULL, col = NULL, i = NULL, ..., tf)

## S3 method for class 'array'
sb_tf(x, idx = NULL, dims = NULL, i = NULL, ..., tf)

## S3 method for class 'data.frame'
sb_tf(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ..., tf)
```

Arguments

x	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.

`...` further arguments passed to or from other methods.

`i` `sb_rp(x, i = i, tf = tf)` corresponds to `x[i] <- tf(x[i])`.
Any of the following can be given here:

- `NULL`, only for multi-dimensional objects or factors. `NULL` results in the entire object being transformed.
- a vector of length 0, in which case the original object is returned unchanged.
- a strictly positive numeric vector with indices (duplicates are NOT allowed).
- logical vector (without `NA`s) of the same length as `x` giving the indices to transform.
- a character vector of index names (duplicates are NOT allowed). If an object has multiple indices with the given name, all the values of the corresponding indices will be transformed.
- function that returns a logical vector giving the element indices to select.

`tf` the transformation function to use.

`row, col` `sb(x, row, col, rp = rp)` corresponds to `x[row, col] <- tf(x[row, col, drop = FALSE])`.
Thus `row` = rows, `col` = columns.
Any of the following can be given here:

- `NULL` (default), which results in ALL of the indices this dimension being transformed.
- a vector of length 0, in which case the original object is returned unchanged.
- a strictly positive integer vector with dimension indices to transform (duplicates are NOT allowed).
- logical vector (without `NA`s) of the same length as the corresponding dimension size, giving the indices of this dimension to transform.
- a character vector of index names (duplicates are NOT allowed). If an object has multiple indices with the given name, all the corresponding indices will be transformed.

NOTE: The arguments `row` and `col` will be ignored if `i` is specified.

`idx, dims` arguments to subset arrays:

- `idx`: a list of indices.
- `dims`: a integer vector of the same length as `idx`, giving the dimensions to which the indices given in `idx` correspond to.

The elements of `idx` follow the same rules as the rules for `row` and `col`, EXCEPT one should not fill in `NULL`.
Thus `sb_tf(x, list(1:10, 1:4), c(1, 3), tf = tf)` is equivalent to `x[1:10, , 1:4] <- tf(x[1:10, , 1:4, drop = FALSE])`.
NOTE: The arguments `idx` and `dims` will be ignored if `i` is specified.

`filter` a one-sided formula with a single logical expression using the column names of the data.frame, giving the condition which observations (rows) should be transformed.

`vars` a function, giving the condition which variables (columns) should be transformed.

Details

One cannot specify `i` and `row/col/lvl/idx/dims` simultaneously. It's either `i`, or the other arguments.

One cannot specify `row` and `filter` simultaneously. It's either one or the other. Similarly, one cannot specify `col` and `vars` simultaneously.

In the above cases it holds that if one set is specified, the other is set is ignored.

Value

A copy of the transformed object.

Examples

```
# vector-like objects ====
obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_tf(obj, 1:3, 1:3, tf = \(x)x^2)
# above is equivalent to obj[1:3, 1:3] <- (obj[1:3, 1:3, drop = FALSE])^2; obj
sb_tf(obj, i = \(x)x>5, tf = \(x)x^2)
# above is equivalent to obj[obj > 5] <- (obj[obj > 5])^2; obj
sb_tf(obj, col = "a", tf = \(x)x^2)
# above is equivalent to obj[, colnames(obj) %in% "a"] <- (obj[, colnames(obj) %in% "a"])^2; obj

obj <- array(1:64, c(4,4,3))
print(obj)
sb_tf(obj, list(1:3, 1:2, c(1, 3)), 1:3, tf = \(x)x^2)
# above is equivalent to obj[1:3, 1:2, c(1, 3)] <- (obj[1:3, 1:2, c(1, 3), drop = FALSE])^2
sb_tf(obj, i = \(x)x<=5, tf = \(x)x^2)
# above is equivalent to obj[obj <= 5] <- (obj[obj <= 5])^2


# lists ====
obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_tf(obj, "a", tf =  \(x)x^2)
# above is equivalent to obj[names(obj) %in% "a"] <- (obj[names(obj) %in% "a"])^2
sb_tf(obj, is.numeric, tf = \(x)x^2)
# above is equivalent to obj[sapply(obj, is.numeric)] <- lapply(obj[sapply(obj, is.numeric)], \(x)x^2); obj


# data.frame-like objects ====
obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb_tf(obj, filter = ~ (a > 5) & (c < 19), vars = is.numeric, tf = \(x)x^2)
```

---

sb_x *Method to extract, exchange, or duplicate indices of an object*

Description

This is an S3 Method to extract, exchange, or duplicate indices of an object.

Usage

```
sb_x(x, ...)

## Default S3 method:
sb_x(x, i, ...)

## S3 method for class 'factor'
sb_x(x, i = NULL, lvl = NULL, drop = FALSE, ...)

## S3 method for class 'list'
sb_x(x, i, drop = FALSE, ...)

## S3 method for class 'matrix'
sb_x(x, row = NULL, col = NULL, i = NULL, ...)

## S3 method for class 'array'
sb_x(x, idx = NULL, dims = NULL, i = NULL, ...)

## S3 method for class 'data.frame'
sb_x(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ...)
```

Arguments

`x`	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.
`...`	further arguments passed to or from other methods.
`i`	`sb_x(x, i = i)` corresponds to `x[i]`. Any of the following can be given here: • `NULL`, only for multi-dimensional objects or factors. `NULL` results in the entire object being returned. • a vector of length 0, in which case an empty object of length 0 is returned. • a strictly positive integer vector with indices (duplicates are allowed, resulting in duplicated indices). • logical vector (without `NA`s) of the same length as `x` giving the indices to extract. • a character vector of index names (duplicates are allowed, resulting in duplicated indices). If an object has multiple indices with the specified name, all the corresponding indices will be extracted/exchanged/duplicated. • function that returns a logical vector giving the element indices to select.
`lvl`	names of the levels of `x`, for which the corresponding indices are to be extracted, exchanged, or duplicated. Duplicates are allowed, resulting in duplicated indices.
`drop`	logical. • For factors: If `TRUE`, unused levels are dropped, if `FALSE` they are not dropped. • For lists: is `TRUE`, selecting a single element will give the simplified result, like using `[[]]`. If `FALSE`, a list is always returned regardless of the number of elements.
`row, col`	`sb(x, row, col)` corresponds to `x[row, col, drop = FALSE]`. Thus `row` = rows, `col` = columns. Any of the following can be given here:

- NULL (default), which results in ALL the indices of this dimension being selected, and the entire object is returned.
- a vector of length 0, in which case an empty object is returned.
- a strictly positive integer vector with dimension indices (duplicates are allowed, resulting in duplicated indices).
- logical vector (without NAs) of the same length as the corresponding dimension size, giving the indices of this dimension to extract.
- a character vector of index names (duplicates are allowed, resulting in duplicated indices). If an object has multiple indices with the given name, all the corresponding indices will be extracted/exchanged/duplicated.

NOTE: The arguments row and col will be ignored if i is specified.

idx, dims — arguments to subset arrays:

- idx: a list of indices.
- dims: a integer vector of the same length as idx, giving the dimensions to which the indices given in idx correspond to.

The elements of idx follow the same rules as the rules for row and col, EXCEPT one should not fill in NULL.
Thus sb_rm(x, list(1:10, 1:4), c(1, 3)) is equivalent to x[1:10, , 1:4, drop = FALSE].
NOTE: The arguments idx and dims will be ignored if i is specified.

filter — a one-sided formula with a single logical expression using the column names of the data.frame, giving the condition which observations (rows) should be extracted.
For example:
to select rows for which column a > 2 and for which column b != "a", specify the following formula:
~ (a > 2) & (b != "a")

vars — a function, giving the condition which variables (columns) should be extracted.

Details

One cannot specify i and row/col/lvl/idx/dims simultaneously. It's either i, or the other arguments.

One cannot specify row and filter simultaneously. It's either one or the other. Similarly, one cannot specify col and vars simultaneously.

In the above cases it holds that if one set is specified, the other is set is ignored.

Value

A copy of the sub-setted object.

Examples

```
# vector-like objects ====
obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_x(obj, 1:3, 1:3)
# above is equivalent to obj[1:3, 1:3, drop = FALSE]
```

```
sb_x(obj, i = \(x)x>5)
# above is equivalent to obj[obj > 5]
sb_x(obj, col = c("a", "a"))
# above is equivalent to obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()]

obj <- array(1:64, c(4,4,3))
print(obj)
sb_x(obj, list(1:3, 1:2, c(1, 3)), 1:3)
# above is equivalent to obj[1:3, 1:2, c(1, 3), drop = FALSE]
sb_x(obj, i = \(x)x>5)
# above is equivalent to obj[obj > 5]


# lists ====
obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_x(obj, 1) # obj[1]
sb_x(obj, 1, drop = TRUE) # obj[[1]]
sb_x(obj, 1:2) # obj[1:2]
sb_x(obj, is.numeric) # obj[sapply(obj, is.numeric)]
# for recusive indexing, see sb_rec()


# factors ====
obj <- factor(rep(letters[1:5], 2))
sb_x(obj, lvl = c("a", "a"))
# above is equivalent to obj[lapply(c("a", "a"), \(i) which(obj == i)) |> unlist()]


# data.frame-like objects ====
obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb_x(obj, 1:3, 1:3) # obj[1:3, 1:3, drop = FALSE]
sb_x(obj, filter = ~ (a > 5) & (c < 19), vars = is.numeric)
```

`seq_names` *Generate Integer Sequence From a Range of Names*

Description

This is an S3 method.

Usage

```
seq_names(names, start, end, inv = FALSE)
```

Arguments

`names`	a character vector of names. Duplicate names, empty names, or a character vector of length zero are not allowed.
`start`	the name giving the starting index of the sequence

end	the name giving the ending index of the sequence
inv	logical, if TRUE, the indices of all names EXCEPT the names of the specified sequence will be given.

Value

An integer vector.

Examples

```
x <- data.frame(a = 1:10, b = letters[1:10], c = factor(letters[1:10]), d = -1:-10)
ind <- seq_names(colnames(x), "b", "d")
sb_x(x, col = ind)
```

---

seq_rec *Recursive Sequence Generator*

Description

This is a recursive sequence generator. The function is essentially a highly generalized version of a Fibonacci sequence generator. One can change the initial values, the window size, and even the window function used.
This function assumes only the following about the sequence being generated:

- The sequence consists of real numbers (i.e. class `integer` or class `double`).
- The window size is the same for all iterations.
- The window function is the same for all iterations.
- The sequence grows until a vector of length `n` is achieved.

Usage

```
seq_rec(inits = c(0, 1), n = 10L, f = sum)
```

Arguments

inits	a numeric (double or integer) vector giving the initial values. Any numbers are allowed, even negative and/or fractional numbers. Note that numbers given must give valid results when passed to function `f()`. IMPORTANT: The length of `inits` determines the window size. For a regular Fibonacci, `inits = 0:1`, which of course means a window size of 2.
n	a single integer, giving the size of the numeric vector to generate. NOTE: it must hold that `n` is larger than or equal to the window size. The window size is equal to `length(inits)`.
f	the function to be used on the last `w` numbers to generate the next number of the sequence at each iteration. This must be a function that takes as input a numeric vector, and returns a single numeric value.

For a regular Fibonacci sequence, this would be either:
`f = sum`,
or (since window size is 2) `f = \(x) x[2] + x[1]`

Details

The default values of the arguments give the first 10 numbers of a regular Fibonacci sequence.
See examples for several number series created with this function.
This function is written in C++ using `Rcpp` for better performance.

Value

A sequence of numbers.

Examples

```
seq_rec() # by default gives Fibonacci numbers
seq_rec(0:3, 10L, sum) # a weird shifted version of Fibonacci
seq_rec(inits=2:1) # Lucas numbers
c(1, seq_rec(c(1, 2), f=prod)) # Multiplicative Fibonacci
seq_rec(f=\(x)2*x[2]+x[1]) # Pell numbers
seq_rec(inits = c(1, 0), f=\(x)2*x[1]) # see https://oeis.org/A077957
seq_rec(f=\(x)x[2]+2*x[1]) # Jacobsthal numbers
seq_rec(c(1,1,1), f=\(x)x[1] + x[2]) # Padovan sequence
seq_rec(c(3,0,2), f=\(x)x[1] + x[2]) # Perrin numbers
seq_rec(c(0,1,3), f=\(x)3*x[3] - 3*x[2] + x[1]) # Triangular numbers
```

Index