# Package 'subsets'

November 12, 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 names and other attributes.
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).
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'.

## Goal & Properties

The Goal of the 'subsets' package is NOT to replace the square-brackets operators, (see Extract), but to provide **alternative** sub-setting methods and functions, to be used in situations where the square-brackets operators are inconvenient.
These are (hopefully) easier sub-setting methods and 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 arguments 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.
  - All functions return a copy of the object, unless stated otherwise.
- **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.
- **Explicit copy semantics**:
  - Sub-set operations that increase or decrease the number of elements in an object, and thus change its memory allocations, always return a modified copy of the object.
  - For sub-set operations that just change values in-place (similar to the `[<-` and `[[<-` methods) the user can choose a method that modify the object by **reference**, or choose a method that return a **deep copy**.
- **Careful handling of names and other attributes**:
  - Sub-setting object by index names returns ALL indices with that name, not just the first.
  - Data.frame-like objects (see supported classes below) are forced to have unique column names.
  - Selecting non-existing names always gives an error.
  - Attributes of data.frame-like objects (see supported classes below) are always preserved when sub-setting.
  - For other object types, the user can specify whether to preserve Attributes, or use R's `[` attribute behaviour (i.e. drop most attributes). This is to ensure compatibility with R packages that create their own attribute behaviour for sub-setting.
- **Support a wide variety of data types**:

  - Support vector-like (atomic) objects (vectors, matrices, arrays).
  - Support lists.
  - Support factors.
  - Support the following data.frame-like objects: data.frame, data.table, tibble, and tidytable class, and objects derived from these classes.
  - Support for the column selection sub-setting used in ggplot2's aes function.
  - Support for sub-setting characters in a single string.
  - Since the main functions are S3 functions, other packages may add functionality for additional classes.
- **Concise function and argument names**.
- **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_mod: method to return a **copy** of an object with modified (transformed or replaced values) subsets.
- sb_set: method to modify (transform or replace values) subsets of an object by **reference**.
- sb_before, sb_after: methods to insert new values before or after an index along a dimension of an object.
- sb_rec: not actually a method, but a function that can be combined with the above methods, for recursive sub-setting operations.

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: extract or replace a subset of characters of a single string (each single character is treated as a single element).
- sb_a: extract multiple attributes from an object.

And finally, a couple of helper functions for creating ranges, sequences, and indices (sometimes 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.
- sub2ind, ind2sub: Convert subscripts (array indices) to flat indices, and vice-versa.

## 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.

---

`aaa1_subsets_indx_args`
*Index Arguments in the Generic Sub-setting Methods*

---

## Description

There are 4 types of arguments that can be used in the generic methods of 'subsets' to specify the indices to perform operations on:

- `i`: to specify flat (i.e. dimensionless) indices.
- `row, col`: to specify rows and/or columns in tabular objects.
- `idx, dims`: to specify indices of arbitrary dimensions in arrays.
- `filter, vars`: to specify rows and/or columns specifically in data.frame-like objects.

## Argument i

class: vector-like
class: factor
class: list

Any of the following can be specified for argument `i`:

- `NULL`, only for multi-dimensional objects or factors, when specifying the other arguments (i.e. dimensional indices or factor levels.)
- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a **strictly positive integer** vector with indices.

- a **logical vector** (without NAs!), of the same length as x, giving the indices to select for the operation.
- a **character** vector of index names. If an object has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.
- a **function** that returns a logical vector, giving the element indices to select for the operation.

Using the i arguments corresponds to doing something like the following:

```
sb_x(x, i = i) ==> x[i]
sb_rm(x, i = i) ==> remove x[i]
sb_mod(x, i = i, rp = rp) ==> x[i] <- rp
sb_mod(x, i = i, tf = tf) ==> x[i] <- tf(x[i])
```

### Arguments row, col

class: matrix
class: data.frame-like

Any of the following can be specified for the arguments row / col:

- NULL (default), corresponds to a missing argument, which results in ALL of the indices in this dimension being selected for the operations.
- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a **strictly positive integer** vector with dimension indices to select for the operation.
- a **logical** vector (without NAs!) of the same length as the corresponding dimension size, giving the indices of this dimension to select for the operation.
- a **character** vector of index names. If an object has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

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

Using the row, col arguments corresponds to doing something like the following:

```
sb_x(x, row = row, col = col) ==> x[row, col, drop = FALSE]
sb_rm(x, row = row, col = col) ==> remove x[row, col, drop = FALSE]
sb_mod(x, row = row, col = col, rp = rp) ==> x[row, col] <- rp
sb_mod(x, row = row, col = col, tf = tf) ==> x[row, col, drop = FALSE] <- tf(x[row, col, drop = FALSE]
```

### Arguments idx, dims

class: array

idx must be a list of indices.
dims must be an 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.
NOTE: The arguments idx and dims will be ignored if i is specified.

Using the idx, dims arguments, corresponds to doing something like the following, here using an example of a 4-dimensional array:

```
sb_mod(x, list(1:10, 1:5), c(1, 3), rp = rp) ==> x[1:10, , 1:5, ] <- rp
sb_mod(x, list(1:10, 1:5), c(1, 3), tf = tf) ==> x[1:10, , 1:5, ] <- tf(x[1:10, , 1:5, , drop = FALSE])
```

### Arguments filter, vars

class: data.frame-like

`filter` must be a one-sided formula with a single logical expression using the column names of the data.frame, giving the condition which observation/row indices should be selected for the operation.
For example, to perform an operation on the rows for which column `height` > `2` and for which column `sex` != `"female"`, specify the following formula:

```
~ (height > 2) & (sex != "female")
```

`vars` must be a function that returns a logical vector, giving the column indices to select for the operation.
For example, to select all numeric columns, specify `vars` = `is.numeric`.

### Argument lvl

class: factor

For this argument, the names of the levels of `x` can be given, selecting the corresponding indices for the operation.

### Duplicates (for Names, Integers, and Levels)

Generally speaking, duplicate names, integers, or levels are NOT allowed in index selection.
The exception is the sb_x method, as that method can be used for duplicating indices.

### Disallowed Combinations of Index Arguments

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.

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 | See subsets_indx_args. |
| 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 = NULL, ...)

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

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

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

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

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

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

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

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

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

## S3 method for class 'data.frame'
sb_after(x, new, margin, pos = collapse::fdim(x)[margin], .attr = 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. |
| new | the new value(s). The type of object depends on x:<br>• For vector-like objects, new can be any vector-like object. However, if one wished the added values in new to be named, 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.)<br>• For factors, new must be a factor.<br>• For lists, new must be a (possible named) list.<br>• 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.<br>By default, concatenation strips attributes, since the attributes of x and new may not be equal or even compatible.<br>In the attr argument, the attributes of the merged object can be specified.<br>Only attributes that are actually missing AFTER insertion will be added, thus preventing overwriting existing attributes like names.<br>One may, for example, specify .attr = sb_a(x) or .attr = sb_a(new).<br>If NULL (the default), no attributes will be added post-insert.<br>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_mod *Method to Return a Copy of an Object With Modified Subsets*

---

## Description

This is an S3 Method to return a copy of an object with modified subsets.

## Usage

```
sb_mod(x, ...)

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

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

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

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

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

## S3 method for class 'data.frame'
sb_mod(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ..., rp, 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, lvl, row, col, idx, dims, filter, vars` | See subsets_indx_args.<br>An empty index selection returns the original object unchanged. |
| `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. |
| `tf` | the transformation function. |

## Details

**Transform or Replace**
Specifying argument `tf` will transform the subset.
Specifying `rp` will replace the subset.
One cannot specify both `tf` and `rp`. It's either one set or the other.
Note that the `tf` argument is not available for factors: this is intentional.

## Value

A copy of the object with replaced/transformed values.

## Examples

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

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_mod(obj, 1:3, 1:3, tf = \(x) -x)
# above is equivalent to  obj[1:3, 1:3] <- -1 * obj[1:3, 1:3]
sb_mod(obj, i = \(x)x<=5, tf = \(x) -x)
```

```
# above is equivalent to  obj[obj <= 5] <- -1:-5; obj
sb_mod(obj, col = "a", tf = \(x) -x)
# above is equivalent to  obj[, which(colnames(obj) %in% "a")] <- -1:-8; obj


obj <- array(1:64, c(4,4,3))
print(obj)
sb_mod(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_mod(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_mod(obj, "a", rp = list(1))
# above is equivalent to  obj[["a"]] <- 1; obj
sb_mod(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_mod(obj, vars = is.numeric, rp = data.frame(-1:-10))
```

---

sb_rec *Access recursive subsets*

---

## Description

The sb_rec() function allows the user to access recursive subsets of lists, and can be combined (i.e. piped) with the generic methods provided by 'subsets'.

## Usage

```
sb_rec(lst, rec)
```

## Arguments

| | |
|---|---|
| lst | a list, or list-like object. |
| 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.<br>When on a certain subset level of a nested list, multiple subsets with the same name exist, only the first one will be selected when performing recursive indexing by name, due to the recursive nature of this type of subsetting. |

## Value

The subsetted object.

## Examples

```
lst <- list(
  A = list(
    A = list(A = "AAA", B = "AAB"),
    A = list(A  = "AA2A", B = "AA2B"),
    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 "AA2B"
sb_rec(lst, c("A", "B", "B")) # this gives "ABB"
sb_rec(lst, c(2,2,1)) # this gives "BBA"
sb_rec(lst, c("B", "B", "A")) # this gives "BBA"

# return a modified copy of the second-lowest level,
# where replace "ABB" is replaced with -1:
sb_rec(lst, c("A", "B")) |> sb_mod(i = "B", rp = list(-1))

# replace "AAA" with -1 BY REFERENCE:
sb_rec(lst, c("A", "A")) |> sb_set(i = "A", rp = list(-1))
lst # notice the first element is replaced by -1
```

---

# sb_rm *Method 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, ..., rat = FALSE)

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

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

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

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

## 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, lvl, row, col, idx, dims, filter, vars` | See subsets_indx_args.<br>An empty index selection results in nothing being removed, and the entire object is returned. |
| `rat` | logical, indicating if attributes should be returned with the sub-setted object. See Details section for more info. |
| `drop` | logical.<br>• For factors: If `drop = TRUE`, unused levels are dropped, if `drop = FALSE` they are not dropped.<br>• For lists: if `drop = TRUE`, selecting a single element will give the simplified result, like using `[[]]`. If `drop = FALSE`, a list is always returned regardless of the number of elements. |

## Details

### One the `rat` argument

Most `[` - methods strip most (but not all) attributes.
If `rat = FALSE`, this default behaviour is preserved, for compatibility with special classes. This is the fastest option.
If `rat = TRUE`, attributes from `x` missing after sub-setting are re-assigned to `x`. Already existing attributes after sub-setting will not be overwritten.
There is no `rat` argument for data.frame-like object: their attributes will always be preserved.
NOTE: In the following situations, the `rat` argument will be ignored, as the attributes necessarily have to be dropped:

- when `x` is a list, AND `drop = TRUE`, AND a single element is selected.
- when `x` is a matrix or array, and sub-setting is done through the `i` argument.

## 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_set *Method to Modify Subsets of an Object By Reference*

---

## Description

This is an S3 Method to replace or transform a subset of an object BY REFERENCE.
WARNING: this method is currently experimental!

## Usage

```
sb_set(x, ...)

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

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

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

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

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

## S3 method for class 'data.frame'
sb_set(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ..., rp, 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, lvl, row, col, idx, dims, filter, vars` | See subsets_indx_args.<br>An empty index selection returns the original object unchanged. |
| `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. |
| `tf` | the transformation function. |

## Details

**Transform or Replace**

Specifying argument `tf` will transform the subset. Specifying `rp` will replace the subset. One cannot specify both `tf` and `rp`. It's either one set or the other.

Note that the `tf` argument is not available for factors: this is intentional.

## Value

Returns: VOID. This method modifies the object by REFERENCE.

Do NOT use assignment like `x <- sb_set(x, ...)`.

Since this function returns void, you'll just get NULL.

## Examples

```
gen_mat <- function() {
  obj <- matrix(1:16, ncol = 4)
  colnames(obj) <- c("a", "b", "c", "a")
  return(obj)
}
# vector-like objects ====
obj <- obj2 <- gen_mat()
obj
sb_set(obj, 1:3, 1:3, rp = -1:-9)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, i = \(x)x<=5, rp = -1:-5)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, col = "a", rp = cbind(-1:-4, -5:-8))
obj2

obj <- obj2 <- gen_mat()
obj
sb_set(obj, 1:3, 1:3, tf = \(x) -x)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, i = \(x)x<=5, tf = \(x) -x)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, col = "a", tf = \(x) -x)
obj2


gen_array <- function() {
  array(1:64, c(4,4,3))
}
obj <- gen_array()
obj
sb_set(obj, list(1:3, 1:2, c(1, 3)), 1:3, rp = -1:-12)
obj
obj <- gen_array()
obj
sb_set(obj, i = \(x)x<=5, rp = -1:-5)
obj
```

---

sb_special *Specialized subsetting functions*

---

## Description

The sb_a() function subsets extracts one or more attributes from an object.
The sb_str() function subsets characters of single string, or replace a subset of the characters of a single string with the subsets of the characters of another string. In both cases, a single string is treated as a iterable vector, where each single character in a string is a single element.

## Usage

```
sb_str(str, ind, rp.str, rp.ind)

sb_a(x, a = NULL)
```

## Arguments

| | |
|---|---|
| str | a single string. |
| ind | an integer vector, giving the positions of the string to subset. |
| rp.str, rp.ind | similar to str and ind, respectively.<br>If not specified, sb_str() will perform something like<br>str[ind]<br>treating str as an iterable vector.<br>If these ARE specified, sb_str() will perform something like<br>str[ind] <- rp.str[ind.str]<br>treating str and rp.str as iterable vectors. |
| x | an object |
| a | a character vector of attribute names. If NULL (default), ALL attributes are returned. |

## Value

The sub-setted object.

## Examples

```
x <- matrix(1:10, ncol = 2)
colnames(x) <- c("a", "b")
attr(x, "test") <- "test"
sb_a(x, "test")
sb_a(x)

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"
sb_str(x, 1:4, "world", 1:4) # this gives "worlo"
```

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, ..., rat = FALSE)

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

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

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

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

## 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, lvl, row, col, idx, dims, filter, vars | See subsets_indx_args.<br>Duplicates are allowed, resulting in duplicated indices.<br>An empty index selection results in an empty object of length 0. |
| rat | logical, indicating if attributes should be returned with the sub-setted object. See Details section for more info. |
| drop | logical.<br>• For factors: If drop = TRUE, unused levels are dropped, if drop = FALSE they are not dropped.<br>• For lists: if drop = TRUE, selecting a single element will give the simplified result, like using [[]]. If drop = FALSE, a list is always returned regardless of the number of elements. |

## Details

### One the `rat` argument

Most `[` - methods strip most (but not all) attributes.
If `rat = FALSE`, this default behaviour is preserved, for compatibility with special classes. This is the fastest option.
If `rat = TRUE`, attributes from `x` missing after sub-setting are re-assigned to `x`. Already existing attributes after sub-setting will not be overwritten.
There is no `rat` argument for data.frame-like object: their attributes will always be preserved.
NOTE: In the following situations, the `rat` argument will be ignored, as the attributes necessarily have to be dropped:

- when `x` is a list, AND `drop = TRUE`, AND a single element is selected.
- when `x` is a matrix or array, and sub-setting is done through the `i` argument.

## Value

Returns 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.<br>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.<br>Any numbers are allowed, even negative and/or fractional numbers.<br>Note that numbers given must give valid results when passed to function `f()`.<br>IMPORTANT: The length of `inits` determines the window size.<br>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.<br>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.<br>This must be a function that takes as input a numeric vector, and returns a single numeric value.<br>For a regular Fibonacci sequence, this would be either:<br>`f = sum`,<br>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
```

---

# sub2ind *Convert Subscripts (Array Indices) to Flat Indices, and Vice-Versa.*

---

## Description

The `sub2ind()` function converts subscripts (i.e. coordinate-like indices) to flat (1D) indices, and the `ind2sub()` does the opposite.
The `sub2ind()` is thus the opposite of arrayInd, and `ind2sub` is merely a convenient wrapper around arrayInd.

Both functions are written to be memory-efficient.

## Usage

```
sub2ind(coords, x.dim, x.len, checks = TRUE)

ind2sub(ind, x.dim, x.len)
```

## Arguments

| | |
|---|---|
| `coords` | an integer matrix, giving the coordinate indices (subscripts) to convert.<br>Each row is an index, and each column is the dimension.<br>The number of columns of `coords` must be equal to the length of `x.dim`. |
| `x.dim` | an integer vector giving the dimensions of the array in question. I.e. `dim(x)`. |
| `x.len` | the length of the object, i.e. `length(x)`. This is needed to evaluate the dimensions. |
| `checks` | logical, indicating if arguments checks should be performed.<br>Defaults to `TRUE`. Can be set to `FALSE` for minor speed improvements, but not recommended. |
| `ind` | an integer vector, giving the flat position indices to convert. |

## Value

The converted indices.

## Examples

```
x.dim <- c(1000, 10, 4, 4)
x.len <- prod(x.dim)
x <- array(1:x.len, x.dim)
x[4,3,2, 1]
x[1,2,3,4]
coords <- rbind(c(4:1), 1:4)
ind <- sub2ind(coords, x.dim, x.len)
print(ind)
x[ind] == c(x[4, 3, 2, 1], x[1, 2, 3, 4]) # TRUE, TRUE
ind2sub(ind, x.dim, x.len)
```

# Index