

Package ‘squarebrackets’

March 12, 2024

Type Package

Title Methods as an Alternative to the Square Brackets Operators

Version 0.0.0.9

Date 2024-02-21

Description Methods as an alternative to the Square Brackets Operators.

These methods have the following properties.

- 1) Programmatically friendly.
- 2) Class consistent.
- 3) Explicit copy semantics.
- 4) Careful handling of names and other attributes.
- 5) Performance aware.
- 6) Supported immutable classes include
atomic, factor, list, (sf-) data.frame/tibble.
- 7) Supported mutable classes include
mutable_atomic, views of lists, and (sf-) data.table/tidytibble.

License MIT + file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests rlang,
knitr,
rmarkdown,
tinytest,
tinycodet,
abind,
tidytibble,
tibble,
ggplot2,
sf,
future.apply,
stringi,
collections,
fastmap

Depends R (>= 4.2.0)

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

URL <https://github.com/tony-aw/squarebrackets/>, <https://tony-aw.github.io/squarebrackets/>

BugReports <https://github.com/tony-aw/squarebrackets/issues/>

Language en-gb

R topics documented:

aaa0_squarebrackets	2
aaa1_squarebrackets_immutable_classes	6
aaa2_squarebrackets_mutable_classes	8
aaa3_squarebrackets_indx_args	12
aaa4_squarebrackets_duplicates	16
aaa5_squarebrackets_PassByReference	17
aaa6_squarebrackets_technical	20
class_mutable_atomic	21
dt	23
idx_by	25
idx_ord_v	27
indx_x	28
match_all	29
n	30
sb_before	30
sb_coe	33
sb_currentBindings	35
sb_mod	37
sb_rec	40
sb_rm	41
sb_set	44
sb_special	47
sb_x	49
seq_mlen	51
seq_names	52
seq_rec	52
setapply	54
sub2ind	55

Index	58
--------------	-----------

aaa0_squarebrackets	<i>squarebrackets: Methods as an Alternative to the Square Brackets Operators</i>
---------------------	---

Description

squarebrackets: Methods as an Alternative to the Square Brackets Operators

Motivation

In order to perform subset operations on some array x with the square brackets operator ($[$, $[-$), one needs to know how many dimensions it has.

I.e. if x has 3 dimensions, one would use

$x[i, j, k, \text{drop} = \text{FALSE}]$ or $x[i, j, k] \leftarrow \text{value}$

But how would one use the square brackets operators, when number of dimensions of x is not known a-priori?

It's not impossible, but still rather convoluted.

The [data.frame](#), [tibble](#), [data.table](#), and [tidytable](#) classes all inherit from class "data.frame".

Yet they use different rules regarding the usage of the square bracket operators.

Constantly switching between these rules is annoying, and makes one's code inconsistent.

When selecting names for sub-setting, only the first occurrences of the names are selected for the sub-set;

and when un-selecting/removing names for sub-setting, the syntax is very different from selecting names.

'R' adheres to copy-on-modification semantics when replacing values using $[-$.

But sometimes one would like explicit control when to create a copy, and when to modify using pass-by-reference semantics.

Among programming languages, 'R' has perhaps one of the most flexible and comprehensive sub-setting functionality, provided by the square brackets operators ($[$, $[-$).

But as shown in the above cases, the square brackets operators are occasionally less than optimally convenient.

There are some packages that solve some of these issues, but I have not found an R-package that provides a holistic approach to providing alternative methods to the square brackets operators.

Thus, this R package was born.

Goal & Properties

The Goal of the 'squarebrackets' package is not to replace the square-brackets operators per-sé (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:**
 - Name-based arguments instead of position-based arguments.
 - Unlike base `[]`, it's not required to know the number of dimensions of an array a-priori, to perform subset-operations on the object.
 - Missing arguments can be filled with `NULL`, instead of using dark magic like `base::quote(expr =)`.
 - No Non-standard evaluation.
 - Functions are pipe-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'.
- **Explicit copy semantics:**
 - Sub-set operations that 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 modifies the object by **reference**, or choose a method that returns a **deep copy**.
- **Careful handling of names and other attributes:**
 - Sub-setting an 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.
 - 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.
- **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. Most of the heavy lifting in this package is done by the 'collapse' package.

Supported Classes

'squarebrackets' only supports S3 classes, and only those that primarily use square brackets for sub-setting (hence the name of the package).

Supported [immutable classes](#):

`atomic`, `factor`, `list`, `data.frame` (including `tibble` and `sf-data.frame`).

Supported [mutable classes](#):

[mutable_atomic](#), `data.table` (including `tidytable` and `sf-data.table`).

No support for mutable list-like classes, such as the various [collections](#) classes from the 'collections' package, and the [fastmap](#) class from the 'fastmap' package, as their sub-setting method is not primarily based on square-brackets.

Methods and Functions

GENERIC METHODS

The main focus is on the following generic methods:

- [sb_x](#): extract, exchange, or duplicate subsets.
- [sb_rm](#): un-select/remove subsets.
- [sb_set](#): modify (transform or replace) subsets of a [mutable object](#) using [Pass By Reference](#) semantics.
- [sb_mod](#): return a **copy** of an object with modified (transformed or replaced) subsets.
- [sb_coe](#): coerce and transform a whole object, or a recursive subset of an object.
- [sb_before](#), [sb_after](#): insert new values before or after an index along a dimension of an object.
- [sb_rec](#): accesses recursive subsets of lists.
- [sb_currentBindings](#): list all currently existing bindings that share the same address as the input variable.

SPECIALIZED FUNCTIONS

Additional specialized sub-setting functions are provided:

- [setapply](#): apply functions over mutable matrix margins using [Pass By Reference](#) semantics.
- [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.
- The [dt_](#)-functions for `data.table`-specific `[]`-operations.

HELPER FUNCTIONS

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

- [n](#): Nested version of [c](#), and short-hand for [list](#).
- Computing indices:
[idx_by](#) to compute grouped indices.
[idx_ord_](#)-functions to compute ordered indices.
- [match_all](#): Find all matches, of one vector in another, taking into account the order and any duplicate values of both vectors.

- Computing sequences:
[seq_rec](#) for the recursive sequence generator (for example to generate a Fibonacci sequence).
[seq_names](#) to create a range of indices from a specified starting and ending name.
[seq_mlen](#) to create multiple sequences of the given lengths at once.
- [sub2coord](#), [coord2ind](#): Convert subscripts (array indices) to coordinates, coordinates to flat indices, and vice-versa.

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

'squarebrackets' relies on the 'collapse', 'Rcpp', 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 sub-setting and data wrangling functionalities.

'squarebrackets' uses a modified version of the [abind](#) function from the 'abind' R-package; the 'abind' package is recommended for binding and sub-filling arrays of arbitrary dimensions.

Besides these package, the following R packages work very nicely together with 'squarebrackets':

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

aaa1_squarebrackets_immutable_classes

Supported Immutable S3 Classes, With Auto-Coercion Rules

Description

The sb_ generic methods support the following categories of S3 classes:

- atomic classes ([vector](#), [matrix](#), [array](#));
- [factor](#);
- [list](#);
- [data.frame](#) (classes that inherit from `data.frame`, like `tibble`, `sf-data.frame` and `sf-tibble`)

Auto-Coercion Rules

Atomic

[coercion_through_copy](#): YES

Atomic objects are automatically coerced to fit the modified subset values, when modifying through copy.

For example, replacing one or multiple values in an integer vector (type `int`) with a decimal number (type `dbl`) will coerce the entire vector to type `dbl`.

Factor

[coercion_through_copy](#): NO

Factors only accept values that are part of their levels, and thus do not support coercion on modification. There is no mechanism for changing factors by reference at all.

Replacing a value with a new value not part of its levels, will result in the replacement value being NA.

List

[coercion_through_copy](#): depends

Lists themselves allow complete change of their elements, since lists are merely pointers.

For example, the following code performs full coercion:

```
x <- list(factor(letters), factor(letters))
sb_mod(x, 1, rp = list(1))
```

However, a recursive subset of a list which itself is not a list, follows the coercion rules of whatever class the recursive subset is.

For example the following code:

```
x <- list(1:10, 1:10)
sb_rec(x, 1, rp = "a") # coerces to character
```

transforms recursive subsets according to the - in this case - atomic auto-coercion rules.

Data.frames when replacing/transforming whole columns

[coercion_through_copy](#): YES

A data.frame is actually a list, where each column is itself a list. As such, replacing/transforming whole columns, so `row = NULL` and `filter = NULL`, allows completely changing the type of the column.

Note that coercion of columns needs arguments `row = NULL` and `filter = NULL` in the [sb_mod](#) and [sb_set](#) methods; NO auto-coercion will take place when specifying something like `row = 1:nrow(x)` (see next section).

Data.frames, when partially replacing/transforming columns

[coercion_through_copy](#): NO

If rows are specified in the [sb_mod](#) and [sb_set](#) methods, and thus not whole columns but parts of columns are replaced or transformed, NO auto-coercion takes place.

I.e.: replacing/transforming a value in an integer (`int`) column to become 1.5, will NOT coerce the column to the decimal type (`dbl`); instead, the replacement value 1.5 is coerced to integer 1.

The `coe` argument in the [sb_mod](#) method allows the user to enforce coercion, even if subsets of

columns are replaced/transformed instead of whole columns.

Specifically, the `coe` arguments allows the user to specify a coercive function to be applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected.

This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`).

Examples

```
# Coercion examples - lists ====
x <- list(factor(letters), factor(letters))
print(x)
sb_mod(x, 1, rp = list(1)) # first element fully changed.

x <- list(1:10, 1:10)
print(x)
sb_rec(x, 1, rp = "a") # coerces first element to character
print(x)

#####

# Coercion examples - data.frame-like - whole columns ====

obj <- data.frame(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

#####

# Coercion examples - data.frame-like - partial columns ====

# sb_mod():
obj <- data.frame(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)

sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  coe = as.double, tf = sqrt # SAFE: coercion performed
)
```

aaa2_squarebrackets_mutable_classes

Supported Mutable S3 classes, With Auto-Coercion Rules

Description

The `sb_` generic methods support the following Mutable S3 classes:

- `mutable_atomic` (this vector class supports any dimension, thus also matrices and arrays);
- `data.table` (and classes that inherit from `data.table`, like `tidytable` and `sf-data.table`);
- **Views of Lists:** Though lists themselves are treated as immutable, lists can contain mutable objects, and so modification by reference of mutable views of lists **is** support by the 'square-brackets'.

The mutable version of the `list` class would be, for example, the various `collections` classes from the 'collections' package, and the `fastmap` class from the 'fastmap' package; both of these use their own sub-setting method which is **not** based on square brackets operators, and thus not covered by this package.

Environments are also not covered by this package.

Auto-Coercion Rules

Coercion Semantics

The mutable classes support "copy-on-modify" semantics like the immutable classes, but - unlike the immutable classes - they also support "pass-by-reference" semantics.

The `sb_mod` method modify subsets of an object through a **deep copy**.

The `sb_set` method and `dt_setcoe` function modify subsets of an object **by reference**.

These 2 copy semantics - "pass by reference" or "modify copy" - have slightly different auto-coercion rules.

These are explained in this section.

Note that the `sb_before` and `sb_after` methods usually allow coercion for all classes.

`mutable_atomic`

`coercion_through_copy`: YES

`coercion_by_reference`: NO

Mutable atomic objects are automatically coerced to fit the modified subset values, when modifying through copy, just like regular atomic classes.

For example, replacing one or multiple values in an integer vector (type `int`) with a decimal number (type `dbl`) will coerce the entire vector to type `dbl`.

Replacing or transforming subsets of mutable atomic objects **by reference** does NOT support coercion. Thus, for example, the following code,

```
x <- 1:16
sb_set(x, i = 1:6, rp = 8.5)
x
```

gives `c(rep(8, 6) 7:16)` instead of `c(rep(8.5, 6), 7:16)`, because `x` is of type integer, so `rp` is interpreted as type integer also.

data.table, when replacing/transforming whole columns

`coercion_through_copy`: YES

`coercion_by_reference`: YES

A `data.table` is actually a list made mutable, where each column is itself a list. As such, replacing/transforming whole columns, so `row = NULL` and `filter = NULL`, allows completely changing the type of the column.

Note that coercion of columns needs arguments `row = NULL` and `filter = NULL` in the `sb_mod` and `sb_set` methods; NO auto-coercion will take place when specifying something like `row = 1:nrow(x)` (see next section).

data.table, when partially replacing/transforming columns

`coercion_through_copy`: NO

`coercion_by_reference`: NO

If rows are specified in the `sb_mod` and `sb_set` methods, and thus not whole columns but parts of columns are replaced or transformed, NO auto-coercion takes place.

I.e.: replacing/transforming a value in an integer (int) column to become 1.5, will NOT coerce the column to the decimal type (dbl); instead, the replacement value 1.5 is coerced to integer 1.

The `coe` argument in the `sb_mod` method allows the user to enforce coercion, even if subsets of columns are replaced/transformed instead of whole columns.

Specifically, the `coe` arguments allows the user to specify a coercive function to be applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected.

This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`).

Views of Lists

`coercion_through_copy`: depends

`coercion_by_reference`: depends

Regular lists themselves are not treated as mutable objects by 'squarebrackets'.

However, lists are not actually really objects, merely a (potentially hierarchical) structure of pointers.

Thus, even if a list itself is not treated as mutable, subsets of a list which are themselves mutable classes, are mutable.

For example, if you have a list of `data.table` objects, the `data.tables` themselves are mutable.

Therefore, the following will work:

```
x <- list(
  a = data.table(cola = 1:10, colb = letters[1:10]),
  b = data.table(cola = 11:20, colb = letters[11:20])
)
mypointer <- x$a
sb_set(mypointer, col = "cola", tf = \(x)x^2)
```

Notice in the above code that `mypointer` is not a copy of `x$a`, since they have the same address.

Thus changing `mypointer` also changes `x$a`.

In other words: `mypointer` is what could be called a "view" of `x$a`.

Notice also that `sb_set(x$a, ...)` will not work, since `sb_set()` requires **actual variables**, similar to in-place functions in the style of ``myfun()`<-``.

The auto-coercion rules of Views of Lists, depends entirely on the object itself. Thus if the list subset is a `data.table`, mutable matrix, coercion rules of `data.tables` apply. And if the list subset is a `data.table`, coercion rules of mutable matrices apply., etc.

Examples

```
# Coercion examples - mutable_atomic ====

x <- as.mutable_atomic(1:16)
sb_set(x, i = 1:6, rp = 8.5) # 8.5 coerced to 8, because `x` is of type `integer`
print(x)

#####

# Coercion examples - data.table - whole columns ====

# sb_mod():
obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

# sb_set():
sb_set(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)
str(obj)

#####

# Coercion examples - data.table - partial columns ====

# sb_mod():
obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)

sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt
  # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
```

```

    coe = as.double, tf = sqrt # SAFE: coercion performed
  )

# sb_set():
obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt
  # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj)
obj <- sb_coe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by sb_coe(); so no warnings
)
print(obj)

#####

# View of List ====

x <- list(
  a = data.table::data.table(col = 1:10, colb = letters[1:10]),
  b = data.table::data.table(col = 11:20, colb = letters[11:20])
)
print(x)
mypointer <- x$a
address(mypointer) == address(x$a) # they are the same
sb_set(mypointer, col = "col", tf = \(x)x^2)
print(x) # notice x has been changed

```

aaa3_squarebrackets_indx_args

Index Arguments in the Generic Sub-setting Methods

Description

There are 6 types of arguments that can be used in the generic methods of 'squarebrackets' 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.
- `rc1`: to specify rows (first dimension), columns (second dimension), and layers (third dimension), in arrays that have exactly 3 dimensions.
- `lvl`: specify levels, for factors only.
- `filter`, `vars`: to specify rows and/or columns specifically in data.frame-like objects.

In this help page, `x` refers to the object to be sub-setted.

Argument `i`

class: atomic
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 takes as input `x`, and returns a logical vector, giving the element indices to select for the operation.
For atomic objects, `i` is interpreted as `i(x)`.
For lists, `i` is interpreted as `lapply(x, i)`.

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

```
sb_x(x, i = i) # ==> x[i]
```

For a brief explanation of the relationship between flat indices (`i`), and the dimension indices (row, col, etc.), see the Details section in [sub2ind](#).

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 operation.
- 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]
```

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.

To keep the syntax short, the user can use the `n` function instead of `list()` to specify idx.

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

```
sb_x(x, n(1:10, 1:5), c(1, 3)) # ==> x[1:10, , 1:5, , drop = FALSE]
```

Arguments rcl

class: array

The rcl argument is only applicable for arrays with exactly 3 dimensions.

If the user knows a-priori that an array has 3 dimensions, using rcl is more efficient than using the idx, dims arguments.

The rcl argument must be a list of exactly 3 elements, with the first element giving the indices of the first dimension (rows), the second element giving the indices of the second dimension (columns), and the third element giving the indices of the third and last dimension (layers); thus rcl stands for "rows, columns, layers" (i.e. the 3 dimensions of a 3-dimensional array).

For each of the aforementioned 3 elements of the list rcl, any of the following can be specified:

- NULL, corresponds to a missing argument, which results in ALL of the indices in this dimension being selected for the operation.

- 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.

By default `rcl` is not a list but simply `NULL`, to be used when specifying the other arguments (either `idx`, `dims` or `i`).

To keep the syntax short, the user can use the `n` function instead of `list()` to specify `rcl`.

Using the `rcl` argument corresponds to doing something like the following:

```
sb_x(x, rcl = n(NULL, 1:10, 1:5)) # ==> x[, 1:10, 1:5, drop = FALSE]
```

Argument `lvl`

class: factor

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

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")
```

If the formula is linked to an environment, any variables not found in the data set will be searched from the environment.

See also `tinycodet::form` for conveniently creating formulas without environments, thus reducing the possibility of memory leaks.

`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`.

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.

Out-of-Bounds Integers and Unknown Names/Levels

Integers that are out of bounds always give an error. Specifying unknown names/levels is considered a form of zero-length indexing.

Disallowed Combinations of Index Arguments

One cannot specify `i` and the other indexing arguments simultaneously; it's either `i`, or the other arguments.

The arguments are evaluated in the following order:

1. Argument `i`
2. Argument `lvl` (for factors) or argument `rcl` (for 3-dimensional arrays)
3. The rest of the indexing 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.

Drop

Sub-setting with the generic methods from the 'squarebrackets' R-package using dimensional arguments (`row`, `col`, `lvl`, `idx`, `dims`, `filter`, `vars`) always use `drop = FALSE`.

To drop potentially redundant (i.e. single level) dimensions, use the `drop` function, like so:

```
sb_x(x, row = row, col = col) |> drop() # ==> x[row, col, drop = TRUE]
```

First, Last, and Shuffle

The indices are counted forward. I.e. 1 is the first element, not the last.

One can use the `last` function to get the last N indices.

One can use the `first` function to get the first N indices.

To shuffle elements of indices, use the `sample` function.

aaa4_squarebrackets_duplicates
On Duplicates

Description

The [sb_x](#) method is the only method where providing duplicate indices actually make sense. For the other methods, it doesn't make sense. Giving duplicate indices usually won't break anything; however, when replacing/transforming or removing subsets, it is almost certainly not the intention to provide duplicate indices. Providing duplicate indices anyway might lead to unexpected results. Therefore, for the methods where giving duplicate indices does not make sense, the `chkdup` argument is present. This argument controls whether the method in question checks for duplicates (TRUE) or not (FALSE).

By default, `chkdup = TRUE`, which means the method in question will check for duplicate indices, and give an error when it finds them.

Setting `chkdup = FALSE` will disable these checks, which saves time and computation power, and is thus more efficient.

Checking for duplicates, is not too expensive, so for the sake of user-friendliness, the default is `chkdup = TRUE`. For optimal performance, please set `chkdup = FALSE`.

aaa5_squarebrackets_PassByReference
Regarding Modification By Reference

Description

This help page describes how modification using "pass-by-reference" semantics is handled by the 'squarebrackets' package.

The main advantage of pass-by-reference is that much less memory is required to modify objects. But at least 2 things should be taken into consideration when modifying an object by reference. First, the coercion rules are slightly different: see [squarebrackets_mutable_classes](#).

Second, if 2 or more variables refer to exactly the same object, changing one variable also changes the other ones.

I.e. the following code,

```
x <- y <- 1:16
sb_set(x, i = 1:6, rp = 8)
```

modifies not just x, but also y.

This is true even if one of the variables is locked (see [bindingIsLocked](#)).

I.e. the following code,

```
tinycodet::import_LL("tinycodet", "%<-c%")
x <- 1:16
y %<-c% x
sb_set(x, i = 1:6, rp = 8)
```

modifies both x and y without error, even though y is a locked constant.

Mutable vs Immutable types

With the exception of environments, most of base R's data types are treated as immutable:

Modifying an object in 'R' will make a copy of the object, something called 'copy-on-modify' semantics.

However, almost any of base R's datatypes can be modified by reference, through R's own 'C' API, or through 'C++' code (like via 'Rcpp'), thus treating these objects as mutable, even though they are not "supposed" to be mutable.

Modifying a base 'R' object by reference can be problematic.

Since 'R', and also most R-packages, treat these objects as immutable, modifying them as-if they are mutable may produce undesired results.

To prevent the issue described above, 'squarebrackets' only supports pass-by-reference semantics on objects that are actually supposed to be mutable.

In relation to this restriction, 'squarebrackets' adds a new class of objects, [mutable_atomic](#), which are simply atomic objects that have the permission to be modified by reference.

Lock Binding

The [lockBinding](#) function locks a binding of an object, preventing modification.

'R' also uses locked bindings to prevent modification of objects from package namespaces.

The pass-by-reference semantics of 'squarebrackets' in principle respect this, and disallows modification of objects by reference.

However, [lockBinding](#) does not lock the address/pointer of an object, only one particular binding of an object.

This problematic; consider the following example:

```
tinycodet::import_LL("tinycodet", "%<-c%")
x <- 1:16
y %<-c% x
sb_set(x, i = 1:6, rp = 8)
```

In the above code, x and y share the same address, thus pointing to the same memory, yet only y is actually locked.

Since x is not locked, modifying x is allowed.

But since `sb_set()` performs modification by reference, y will STILL be modified, despite being locked.

To remedy the issue as explained above, 'squarebrackets' provides the [sb_currentBindings](#) method. This method can do multiple things. One of which is to look at the address of some object x, find all

currently existing bindings in the **caller environment** sharing the same address as `x`, and locking all these bindings.

Warning:

The `sb_currentBindings` method only locks **currently existing** bindings in the **caller environment**; bindings that are created **after** calling `sb_currentBindings` will not automatically be locked.

Thus, every time the user creates a new binding of the same object, and the user wishes it to be locked, `sb_currentBindings` must be called again.

Protected Addresses

To prevent an accidental pass-by-reference modification of objects in the base environment, all addresses of all exported objects in the base environment (`baseenv`) are stored in the option `squarebrackets.protected` whenever 'squarebrackets' is **loaded**, either directly or indirectly.

Needless to say, the user should never touch this option.

Input Variable

Methods/functions that perform in-place modification by reference can be thought of as similar to functions in the style of `some_function(x, ...) <- value`, in the sense that the variable **must actually exist as an actual variable**.

Thus things like any of the following,

`sb_set(1:10, ...)`, `sb_set(x$a, ...)`, or `sb_set(base::letters)`,
will not work.

Views of Lists

Regular lists themselves are not treated as mutable objects by 'squarebrackets'.

However, lists are not actually really objects, merely a (potentially hierarchical) structure of pointers.

Thus, even if a list itself is not treated as mutable, subsets of a list which are themselves mutable classes, are mutable.

For example, if you have a list of `data.table` objects, the `data.tables` themselves are mutable.

Therefore, the following will work:

```
x <- list(
  a = data.table(cola = 1:10, colb = letters[1:10]),
  b = data.table(cola = 11:20, colb = letters[11:20])
)
mypointer <- x$a
sb_set(mypointer, col = "cola", tf = \(x)x^2)
```

Notice in the above code that `mypointer` is not a copy of `x$a`, since they have the same address.

Thus changing `mypointer` also changes `x$a`.

In other words: `mypointer` is what could be called a "view" of `x$a`.

Protection

Due to the properties given in this help page so far, something like the following will not work:

```
# letters = base::letters
sb_set(letters, i = 1, rp = "XXX")
```

The above won't work because:

1. base objects are disallowed;
2. immutable objects are disallowed (you'll have to create a mutable object, which will create a copy of the original, thus keeping the original object safe from modification by reference);
3. locked bindings are disallowed.

Despite the checks made by this package, the user should never actively try to modify a **locked** or **protected** object by reference, as that would defeat the purpose of locking an object.

Examples

```
# the following code demonstrates how locked bindings,
# such as `base::letters`,
# are being safe-guarded

x <- list(a = base::letters)
mypointer <- x$a # view of a list
address(mypointer) == address(base::letters) # TRUE: point to the same memory
bindingIsLocked("letters", baseenv()) # base::letters is locked ...
bindingIsLocked("mypointer", environment()) # ... but this pointer is not!

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    sb_set(mypointer, i = 1, rp = "XXX") # this still gives an error though ...
  )
}

is.mutable_atomic(mypointer) # ... because it's not of class `mutable_atomic`

x <- list(
  a = as.mutable_atomic(base::letters) # `as.mutable_atomic()` makes a copy
)
mypointer <- x$a # view of a list
address(mypointer) == address(base::letters) # FALSE: it's a copy
sb_set(
  mypointer, i = 1, rp = "XXX" # modifies x, does NOT modify `base::letters`
)
```

```
print(x) # x is modified
base::letters # but this still the same

# Word of warning:
# the safe-guard in 'squarebrackets' is good, but definitely not perfect.
# Do not actively try to break things; you might actually succeed.
```

aaa6_squarebrackets_technical

Additional Technical Details

Description

Atomic

[require unique names: NO](#)

The atomic vector is the most basic class type.

Matrices and Arrays are just atomic vectors with dimension attributes.

All elements in an atomic vector class must have the same atomic type ("logical", "integer", "numeric", "complex", "character" and "raw").

Therefore, the [sb_coe](#) method will coerce the entirety of an atomic vector, NOT just a subset of it.

Factor

[require unique names: NO](#)

Factors have the property that they can only store values that are defined in their "levels" attribute; other values are not allowed, and thus result in NAs.

Thus And [sb_mod](#) does NOT coerce replacement values for factors! This is quite different from the atomic vector classes, which can store an infinite variety of values (provided they are of the same atomic type).

Due to these properties, there is NO `sb_set()` method for factors.

List

[require unique names: NO](#)

Lists are recursive objects (i.e. they can be nested), and they do not actually store values but rather store reference to other objects.

Therefore [sb_rec](#) method can be used to access recursive subsets of a list, no matter how deep/low in hierarchy it is in the list.

Data.frame-like

[require unique names: YES](#)

The data.frame-like objects quite different from the previously named classes.

And the different data.frame-like classes also differ from each other quite a bit - especially in terms of sub-setting.

The 'squarebrackets' R-package attempts to keep the data.frame methods as class agnostic as possible, through the class agnostic functionality of the 'collapse' and 'data.table' R-packages.

These 3 things cause some **important** oddities in how data.frame-like classes are treated differently from the other classes:

- Whole-columns will be auto-coerced when replaced/transformed by `sb_mod()`, but partial columns will NOT be auto-coerced.

- The `sb_x` and `sb_rm` methods always automatically conserve all attributes (though names are adjusted accordingly, of course), they are never stripped, unlike the other classes.
- Giving a data.frame-like object with non-unique column names to the `sb_`-methods returns an error. Also, duplicating columns with `sb_x` will automatically adjust the column names to make them unique.

class_mutable_atomic *Mutable Atomic Classes*

Description

The `mutable_atomic` class is a mutable version of atomic classes. It works exactly the same in all aspects as regular atomic classes, with only one real difference: `sb_set` accepts `mutable_atomic`, but does not accept regular `atomic`. See [squarebrackets_PassByReference](#) for details.

Like `data.table`, `[<-` performs R's default copy-on-modification semantics. For modification by reference, use `sb_set`.

The `is.mutable_atomic()` function checks if an object is atomic.

Usage

```
mutable_atomic(data, names = NULL, dim = NULL, dimnames = NULL)

as.mutable_atomic(x, ...)

is.mutable_atomic(x)

## S3 method for class 'mutable_atomic'
x[...]

## S3 replacement method for class 'mutable_atomic'
x[...] <- value

## S3 method for class 'mutable_atomic'
format(x, ...)

## S3 method for class 'mutable_atomic'
print(x, ...)
```

Arguments

<code>data</code>	atomic vector giving data to fill the <code>mutable_atomic</code> object.
<code>names, dim, dimnames</code>	see setNames and array .
<code>x</code>	an atomic object.
<code>...</code>	method dependent arguments.
<code>value</code>	see Extract .

Value

`as.mutable_atomic` converts an atomic object (vector, matrix, array) to the same object, but with additional class `"mutable_atomic"`, and the additional attribute `"typeof"`.

`is.mutable_atomic` returns TRUE if the object is atomic, has the class `"mutable_atomic"`, **and** has the correctly set attribute `"typeof"`;

`is.mutable_atomic` returns FALSE otherwise.

Warning

Always use `mutable_atomic()` or `as.mutable_atomic` to create a mutable object.

Do NOT attempt to manually create a mutable atomic object by tweaking attributes:

`mutable_atomic()` and `as.mutable_atomic` make some necessary checks so that other functions, such as [sb_set](#), don't have to spend computation time to perform these checks also.

Circumventing these checks may break things.

Examples

```
x <- mutable_atomic(
  1:20, dim = c(5, 4), dimnames = list(letters[1:5], letters[1:4])
)
x

x <- matrix(1:10, ncol = 2)
x <- as.mutable_atomic(x)
is.mutable_atomic(x)
print(x)
x[, 1]
x[] <- as.double(x) # notifies the user a copy is being made
print(x) # "typeof" attribute adjusted accordingly, and class still present
```

Description

Functional forms of special `data.table` operations - ALL programmatically friendly (no Non-Standard Evaluation).

`dt_aggregate()` aggregates a `data.table` or `tidytable`, and returns the aggregated copy.

`dt_setcoe()` coercively transforms columns of a `data.table` or `tidytable` using [pass-by-reference semantics](#).

`dt_setrm()` removes columns of a `data.table` or `tidytable` using [pass-by-reference semantics](#).

`dt_setadd(x, new)` adds the columns from `data.table` `new` to `data.table` `x`, thereby modifying `x` using [pass-by-reference semantics](#).

Usage

```
dt_aggregate(x, SDcols = NULL, f, by, order_by = FALSE)

dt_setcoe(x, col = NULL, vars = NULL, f, chkdup = TRUE)

dt_setrm(x, col = NULL, vars = NULL, chkdup = TRUE)

dt_setadd(x, new)
```

Arguments

<code>x</code>	a <code>data.table</code> or <code>tidytable</code> .
<code>SDcols</code>	atomic vector, giving the columns to which the aggregation function <code>f()</code> is to be applied on.
<code>f</code>	the aggregation function
<code>by</code>	atomic vector, giving the grouping columns.
<code>order_by</code>	logical (TRUE or FALSE), indicating if the aggregated result should be ordered by the columns specified in <code>by</code> .
<code>col, vars</code>	columns to select for coercion; see squarebrackets_indx_args . Duplicates are not allowed.
<code>chkdup</code>	see squarebrackets_duplicates .
<code>new</code>	a <code>data.table</code> or <code>tidytable</code> . It must have column names that do not already exist in <code>x</code> .

Value

For `dt_aggregate()`:
The aggregated `data.table` object.

For the rest of the functions:
Returns: VOID. These functions modify the object by reference.
Do NOT use assignment like `x <- dt_setcoe(x, ...)`.
Since these functions return void, you'll just get NULL.

Examples

```
requireNamespace("sf")

# dt_aggregate on sf-data.table ====

x <- sf::st_read(system.file("shape/nc.shp", package = "sf"))
x <- data.table::as.data.table(x)

x$region <- ifelse(x$CNTY_ID <= 2000, 'high', 'low')
d.aggr <- dt_aggregate(
  x, SDcols = "geometry", f= sf::st_union, by = "region"
)

head(d.aggr)
```



```
#####

# dt_setcoe ====

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
str(obj)
obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
dt_setcoe(obj, vars = is.numeric, f = as.numeric) # integers are now numeric
str(obj)
sb_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed; so no warnings
)
str(obj)

#####

# dt_setrm ====

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj)
dt_setrm(obj, col = 1)
str(obj)

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj)
dt_setrm(obj, vars = is.numeric)
str(obj)

#####

# dt_setadd ====

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
new <- data.table::data.table(
  e = sample(c(TRUE, FALSE), 10, TRUE),
  f = sample(c(TRUE, FALSE), 10, TRUE)
)
dt_setadd(obj, new)
```

```
print(obj)
```

idx_by

Compute Grouped Indices

Description

Given:

- a sub-set function `f`;
- the complete range of indices `r` of some object `x`;
- and a grouping factor `grp`;

the `idx_by()` function takes indices `f(r)` **per group** `grp`.

The result of `idx_by()` can be supplied to the indexing arguments (see [squarebrackets_idx_args](#)) of:

[sb_x](#), [sb_rm](#), [sb_mod](#), [sb_set](#), or [sb_coe](#),

to perform **grouped** subset operations.

Usage

```
idx_by(f, r, grp, parallel = FALSE, mc.cores = 1L)
```

Arguments

- | | |
|---|--|
| <code>f</code> | a subset function to be applied per group on <code>r</code> .
The function must accept a character or integer vector as input, and produce a character or integer vector as output.
For example, to subset the last element per group, specify:
<code>f = last</code> |
| <code>r</code> | an integer or character vector, giving the complete range of indices of an object.
For example: <code>colnames(x)</code> , <code>1:nrow(x)</code> , etc. |
| <code>grp</code> | a factor giving the groups. Make sure its order corresponds to <code>i</code> and <code>r</code> , otherwise it makes no sense. |
| <code>parallel</code> , <code>mc.cores</code> | see BY . |

Value

A vector of indices of the same type as `r`.

Examples

```
# vectors ====
(a <- 1:20)
(grp <- factor(rep(letters[1:5], each = 4)))

# get the last element of `a` for each group in `grp`:
i <- idx_by(last, seq_along(a), grp)
sb_x(cbind(a, grp), row = i)

# data.frame ====
x <- data.frame(
  a = sample(1:20),
  b = letters[1:20],
  group = factor(rep(letters[1:5], each = 4))
)
print(x)
# get the first row for each group in data.frame `x`:
row <- idx_by(first, 1:nrow(x), x$group)
sb_x(x, row)
# get the first row for each group for which a > 10:
x2 <- sb_x(x, filter = ~ a > 10)
row <- na.omit(idx_by(first, 1:nrow(x2), x2$group))
sb_x(x2, row)
```

idx_ord_v

Compute Ordered Indices

Description

Computes ordered indices. Similar to [order](#), except the user must supply a vector, a list of equal-length vectors, a data.frame or a matrix (row-wise and column-wise are both supported), as the input.

For a vector `x`, `idx_ord_v(x)` is equivalent to `order(x)`.

For a data.frame or a list of equal-length vectors `x`, with `p` columns/elements, `idx_ord_df(x)` is equivalent to `order(x[[1]], ..., x[[p]])`.

For a matrix (or array) `x` with `p` rows, `idx_ord_m(x, margin = 1)` is equivalent to `order(x[1,], ..., x[p,], ...)`.

For a matrix (or array) `x` with `p` columns, `idx_ord_m(x, margin = 2)` is equivalent to `order(x[, 1], ..., x[, p], ...)`.

Note that these are merely a convenience functions, and that these are actually slightly slower than [order](#) (except for `idx_ord_v()`), due to the additional functionality.

Usage

```
idx_ord_v(  
  x,  
  na.last = TRUE,  
  decr = FALSE,  
  method = c("auto", "shell", "radix")  
)  
  
idx_ord_m(  
  x,  
  margin,  
  na.last = TRUE,  
  decr = FALSE,  
  method = c("auto", "shell", "radix")  
)  
  
idx_ord_df(  
  x,  
  na.last = TRUE,  
  decr = FALSE,  
  method = c("auto", "shell", "radix")  
)
```

Arguments

x	a vector, data.frame, or array
na.last, method	see order and sort .
decr	see argument decreasing in order
margin	the margin over which to cut the matrix/array into vectors. I.e. margin = 1 will cut x into individual rows, and apply the order on those rows. And margin = 2 will cut x into columns, etc.

Value

See [order](#).

Examples

```
x <- sample(1:10)  
order(x)  
idx_ord_v(x)  
idx_ord_m(rbind(x, x), 1)  
idx_ord_m(cbind(x, x), 2)  
idx_ord_df(data.frame(x, x))
```

indx_x	<i>Exported Utilities</i>
--------	---------------------------

Description

Exported utilities

Usage

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

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

Arguments

i	See squarebrackets_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))
```

match_all	<i>Match All, Order-Sensitive and Duplicates-Sensitive</i>
-----------	--

Description

Find all indices of vector haystack that are equal to vector needles, taking into account the order of both vectors, and their duplicate values.

It is essentially a much more efficient version of:

```
lapply(needles, \(i) which(haystack == i))
```

Like `lapply(needles, \(i) which(haystack == i))`, NAs are ignored.

Core of the code is based on a suggestion by Sebastian Kranz (author of the 'collapse' package).

Usage

```
match_all(needles, haystack, unlist = TRUE)
```

Arguments

```
needles, haystack      vectors
unlist                 logical, indicating if the result should be a vector (TRUE, default), or a list (FALSE).
```

Value

An integer vector, or list of integer vector.
If a list, each element of the list corresponds to each value of needles.
When needles and/or haystack is/are empty or fully NA, match_all() returns an empty integer vector (if unlist = TRUE), or an empty list (if unlist = FALSE).

Examples

```
n <- 200
haystack <- sample(letters, n, TRUE)
needles <- sample(letters, n/2, TRUE)
indices1 <- match_all(needles, haystack)
head(indices1)
```

<i>n</i>	<i>Nest</i>
----------	-------------

Description

The `c()` function concatenates vectors or lists into a vector (if possible) or else a list.
In analogy to that function, the `n()` function **ne**sts objects into a list (not into an atomic vector, as atomic vectors cannot be nested).
It is a short-hand version of the `list` function.
This is handy because lists are often needed in 'squarebrackets', especially for arrays.

Usage

```
n()
```

Value

The list.

Examples

```
obj <- array(1:64, c(4,4,3))
print(obj)
sb_x(obj, n(1:3, 1:2), c(1,3))
# above is equivalent to obj[1:3, , 1:2, drop = FALSE]
```

sb_before	<i>Methods to Insert New Values Before or After an Index Along a Dimension</i>
-----------	--

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.

These functions use a modified version of `abind::abind` (see reference below).

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 '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 '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 '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	see squarebrackets_immutable_classes and squarebrackets_mutable_classes .
...	further arguments passed to or from other methods.
new	the new value(s). The type of object depends on x: <ul style="list-style-type: none"> • For atomic objects, new can be any atomic object. However, if one wished the added values in new to be named, ensure new is the same type of object as x. For example: 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, concatenation strips attributes, since the attributes of x and new may not be equal or even compatible. 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. One may, for example, specify .attr = sb_a(x) or .attr = sb_a(new). If NULL (the default), 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.

References

Plate T, Heiberger R (2016). *abind: Combine Multidimensional Arrays*. R package version 1.4-5, <https://CRAN.R-project.org/package=abind>.

Examples

```
# atomic 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)

#####

# factors ====

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

#####

# lists ====

x <- as.list(1:5)
new <- lapply(x, \(x)x*-1)
print(x)
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_coe

Method to Coerce and Transform (Recursive Subsets of) an Object

Description

This is an S3 Method to completely transform (a recursive subsets of) an object with explicit coercion.

Given some coercing function `v()`, the following can be stated about this method.

(1) For atomic objects (vectors, matrices, arrays), this method is equivalent to:

```
x[] <- v(x)
```

(2) For factors, this method is equivalent to:

```
x <- v(x)
```

(3) For lists, with one or multiple elements specified by argument `i`, this method is equivalent to:

```
\(x, i) { x[i] <- lapply(x[i], v); return(x) }
```

(4) And for `data.frame`-like objects, with one or multiple columns specified by argument `col`, this method is equivalent to:

```
collapse::ftransformv(x, col, v)
```

Note that when `x` is a `data.table`, one can coercively transform columns by reference (which is more memory efficient), using [dt_setcoe](#).

Usage

```
sb_coe(x, ...)

## Default S3 method:
sb_coe(x, v, ...)

## S3 method for class 'factor'
sb_coe(x, v, ...)

## S3 method for class 'list'
sb_coe(x, i, v, ...)

## S3 method for class 'data.frame'
sb_coe(x, col = NULL, vars = NULL, v, ...)
```

Arguments

<code>x</code>	see squarebrackets_immutable_classes and squarebrackets_mutable_classes .
<code>...</code>	further arguments passed to or from other methods.
<code>v</code>	the coercive transformation function to use.
<code>i, col, vars</code>	See squarebrackets_indx_args . An empty index selection returns the original object unchanged.

Details

When replacing values by reference, the (recursive subset of the) object is never coerced, as that requires making a deep copy; instead, the replacement value is coerced.

For example:

Using `sb_set()` to replacing/transform one or more values of an integer type (`int`) [mutable_atomic](#) object / [data.table](#) column, to become `1.5`, will NOT coerce the object or column to a decimal type (`dbl`); instead, the replacement `1.5` is coerced to the integer `1`.

For this reason, the `sb_coe()` method can be used to coercively transform an object before replacing or transforming values by reference. See also the Examples section below.

Value

A copy of the coercively transformed object.

Examples

```
obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)
obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
obj <- sb_coe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed; so no warnings
)
print(obj)
```

sb_currentBindings	<i>List or Lock All Currently Existing Bindings Pointing To Same Address</i>
--------------------	--

Description

`sb_currentBindings(x, action = "list")`
 lists all **currently existing** objects sharing the same **address** as `x`, in a given environment.

`sb_currentBindings(x, action = "checklock")`
 searches all **currently existing** objects sharing the same **address** as `x`, in a given environment, and reports which of these are locked and which are not locked.

`sb_currentBindings(x, action = "lockbindings")`
 searches all **currently existing** objects sharing the same **address** as `x`, in a given environment, and locks them using [lockBinding](#).

See also [squarebrackets_PassByReference](#) for information regarding the relation between locked bindings and pass-by-reference modifications.

Usage

```
sb_currentBindings(x, action = "list", env = NULL)
```

Arguments

x	the existing variable whose address to use when searching for bindings.
action	a single string, giving the action to perform. Must be one of the following: <ul style="list-style-type: none"> • "list" (default). • "checklock". • "lockbindings".
env	the environment where to look for objects. If NULL (default), the caller environment is used.

Value

For `sb_currentBindings(x, action = "list")`:
Returns a character vector.

For `sb_currentBindings(x, action = "checklock")`:
Returns a named logical vector.
The names give the names of the bindings,
and each associated value indicates whether the binding is locked (TRUE) or not locked (FALSE).

For `sb_currentBindings(x, action = "lockbindings")`:
Returns VOID. It just locks the currently existing bindings.
To unlock the bindings, remove the objects (see [rm](#)).

Examples

```
x <- as.mutable_atomic(1:10)
y <- x
lockBinding("y", environment())
sb_currentBindings(x)
sb_currentBindings(x, "checklock") # only y is locked

# since only y is locked, we can still modify y through x by reference:
sb_set(x, i = 1, rp = -1)
print(y) # modified!
rm(list= c("y")) # clean up

# one can fix this by locking ALL bindings:
y <- x
sb_currentBindings(x, "lockbindings") # lock all
sb_currentBindings(x, "checklock") # all bindings are locked, including y
# the 'squarebrackets' package respects the lock of a binding,
# provided all bindings of an address are locked;
# so this will give an error, as it should:

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    sb_set(x, i = 1, rp = -1),
```

```

    pattern = "object is locked"
  )
}

# creating a new variable will NOT automatically be locked:
z <- y # new variable; will not be locked!
sb_currentBindings(x, "checklock") # z is not locked
sb_currentBindings(x, "lockbindings") # we must re-run this
sb_currentBindings(x, "checklock") # now z is also locked

if(requireNamespace("tinytest")) {
  tinytest::expect_error( # now z is also protected
    sb_set(z, i = 1, rp = -1),
    pattern = "object is locked"
  )
}

rm(list= c("x", "y", "z")) # clean up

```

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, chkdup = TRUE)

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

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

## S3 method for class 'factor'

```

```

sb_mod(x, i = NULL, lvl = NULL, ..., rp, chkdup = TRUE)

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

## S3 method for class 'data.frame'
sb_mod(
  x,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  coe = NULL,
  ...,
  rp,
  tf,
  chkdup = TRUE,
  .lapply = lapply
)

```

Arguments

x	see squarebrackets_immutable_classes and squarebrackets_mutable_classes .
...	further arguments passed to or from other methods.
i, lvl, row, col, idx, dims, rcl, filter, vars	See squarebrackets_indx_args . 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.
chkdup	see squarebrackets_duplicates .
.lapply	sb_mod() by default uses lapply for lists and dapply data.frame-like objects to compute tf() on every list element or data.frame column. The user may supply a custom lapply()/dapply()-like function in this argument to use instead. For example, to perform parallel transformation, the user may supply <code>future::future_lapply</code> . The supplied function must use the exact same argument convention as lapply , otherwise errors or unexpected behaviour may occur.
coe	For data.frame-like objects, sb_mod() can only coerce whole columns, not subsets of columns. So it does not automatically coerce column types when row or filter is also specified. Therefore, the user can specify a coercion function, to be applied on the entirety of every column specified in col or vars; columns outside this subset are not affected. This coercion function is, of course, applied before replacement (rp) or transformation (tf()). By default, coe = NULL which means no columns are coercively transformed. See also sb_coe .

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

```
# atomic objects ====

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
rp <- -1:-9
sb_mod(obj, 1:3, 1:3, rp = rp)
# 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
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]); obj
sb_mod(obj, i = \ (x)x<=5, tf = \ (x) -x)
# above is equivalent to obj[obj <= 5] <- (-1 * obj[obj <= 5]); 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), rp = -1:-24)
# above is equivalent to obj[1:3, , 1:2] <- -1:-24
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(1L))
# above is equivalent to obj[["a"]] <- 1L; 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]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)

sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
sb_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  coe = as.double, tf = sqrt # SAFE: coercion performed
)

obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

```

sb_rec

Access Recursive Subsets

Description

The `sb_rec()` method allows the user to access recursive subsets of lists.

The `sb_rec()` method also allows replacing or transforming a recursive subset of a list, using R's default in-place semantics, by specifying the `rp` argument.

Usage

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

Arguments

<code>lst</code>	a list, or list-like object.
<code>rec</code>	a vector of length <code>p</code> , such that <code>lst[[rec]]</code> is equivalent to <code>lst[[rec[1]]]</code> ... <code>[[rec[p]]]</code> , providing all but the final indexing results in a list. 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.

rp optional. If specified, performs `lst[[rec]] <- rp`, using R's default in-place semantics.

Value

If `rp` is not specified: Returns the recursive subset.

If `rp` is specified: Returns nothing, but replaces the recursive subset with `rp`, using R's default in-place semantics.

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")
  )
)

#####

# access recursive subsets ====

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"

#####

# replace with R's default in-place semantics ====

# replace "AAB" using R's default in-place semantics:
sb_rec(
  lst, c("A", "A", "B"),
  rp = "THIS IS REPLACED WITH IN-PLACE SEMANTICS"
)
print(lst)

#####

# Modify View of List By Reference ====

x <- list(
  a = data.table::data.table(cola = 1:10, colb = letters[1:10]),
```

```

b = data.table::data.table(cola = 11:20, colb = letters[11:20])
)
print(x)
mypointer <- sb_rec(x, "a")
address(mypointer) == address(x$a) # they are the same
sb_set(mypointer, col = "cola", tf = \ (x)x^2)
print(x) # notice x has been changed

```

sb_rm

Method to Un-Select/Remove Subsets of an Object

Description

This is an S3 Method to un-select/remove subsets from an object.

Usage

```

sb_rm(x, ...)

## Default S3 method:
sb_rm(x, i, ..., rat = FALSE, chkdup = TRUE)

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

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

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

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

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

```

```

    ...,
    chkdup = TRUE
  )

```

Arguments

x	see squarebrackets_immutable_classes and squarebrackets_mutable_classes .
...	further arguments passed to or from other methods.
i, lvl, row, col, idx, dims, rcl, filter, vars	See squarebrackets_indx_args . An empty index selection results in nothing being removed, and the entire object is returned.
rat	TRUE or FALSE, indicating if attributes should be returned with the sub-setted object. See Details section for more info.
chkdup	see squarebrackets_duplicates .
drop	logical. <ul style="list-style-type: none"> • For factors: If drop = TRUE, unused levels are dropped, if drop = FALSE they are not dropped. • For lists: if drop = TRUE, selecting a single element will give the simplified result, like using <code>[[]]</code>. 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`, `chkdup = TRUE`, 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

```

# atomic 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, n(1, c(1, 3)), c(1, 3))
sb_rm(obj, rcl = n(1, NULL, c(1, 3)))
# above 2 lines are equivalent to obj[-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 recursive 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 a Mutable Object By Reference*

Description

This is an S3 Method to replace or transform a subset of a [supported mutable object](#) using [pass-by-reference](#) semantics.

Usage

```
sb_set(x, ...)  
  
## Default S3 method:  
sb_set(x, i, ..., rp, tf, chkdup = TRUE)  
  
## S3 method for class 'matrix'  
sb_set(x, row = NULL, col = NULL, i = NULL, ..., rp, tf, chkdup = TRUE)  
  
## S3 method for class 'array'  
sb_set(  
  x,  
  idx = NULL,  
  dims = NULL,  
  rcl = NULL,  
  i = NULL,  
  ...,  
  rp,  
  tf,  
  chkdup = TRUE  
)  
  
## S3 method for class 'data.table'  
sb_set(  
  x,  
  row = NULL,  
  col = NULL,  
  filter = NULL,  
  vars = NULL,  
  ...,  
  rp,  
  tf,  
  chkdup = TRUE,  
  .lapply = lapply  
)
```

Arguments

x a **variable** belonging to one of the [supported mutable classes](#).

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

i, row, col, idx, dims, rcl, filter, vars
See [squarebrackets_idx_args](#).
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.

chkdup see [squarebrackets_duplicates](#).

.lapply sb_set() by default uses [lapply](#) for lists and [dapply](#) data.frame-like objects to compute tf() on every list element or data.frame column.
The user may supply a custom lapply()/dapply()-like function in this argument to use instead.
For example, to perform parallel transformation, the user may supply `future.apply::future_lapply`.
The supplied function must use the exact same argument convention as [lapply](#), otherwise errors or unexpected behaviour may occur.

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 there is no sb_set() method 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

```
# mutable_atomic objects ====

gen_mat <- function() {
  obj <- as.mutable_atomic(matrix(1:16, ncol = 4))
  colnames(obj) <- c("a", "b", "c", "a")
  return(obj)
}

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() {
  as.mutable_atomic(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

#####

# data.table ====

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
obj <- sb_coe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by sb_coe(); so no warnings
)
print(obj)

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb_set(
  obj, vars = is.numeric,

```

```

    tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
  )
  str(obj)

```

sb_special

Specialized Sub-setting 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. The `sb_str()` function is considerably faster than doing the equivalent operation in base 'R' or even 'stringi'.

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.
If not specified, <code>sb_str()</code> will perform something like
<code>str[ind]</code>
treating str as an iterable vector.
If these ARE specified, <code>sb_str()</code> will perform something like
<code>str[ind] <- rp.str[rp.ind]</code>
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 Subsets of an Object

Description

This is an S3 Method to extract, exchange, or duplicate (i.e. replicate indices x times) indices of an object.

Usage

```
sb_x(x, ...)

## Default S3 method:
sb_x(x, i, ..., 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, rcl = NULL, i = NULL, ..., 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 'data.frame'
sb_x(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ...)
```

Arguments

x see [squarebrackets_immutable_classes](#) and [squarebrackets_mutable_classes](#).
 ... further arguments passed to or from other methods.

i, lvl, row, col, idx, dims, rcl, filter, vars	<p>See squarebrackets_idx_args.</p> <p>Duplicates are allowed, resulting in duplicated indices.</p> <p>An empty index selection results in an empty object of length 0.</p>
rat	logical, indicating if attributes should be returned with the sub-setted object. See Details section for more info.
drop	<p>logical.</p> <ul style="list-style-type: none"> • For factors: If drop = TRUE, unused levels are dropped, if drop = FALSE they are not dropped. • For lists: if drop = TRUE, selecting a single element will give the simplified result, like using <code>[[]]</code>. 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

```
# atomic 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, n(1:3, 1:2), c(1,3))
sb_x(obj, rcl = n(1:3, NULL, 1:2))
# above 2 lines are equivalent to obj[1:3, , 1:2, 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 recursive 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_mlen

Multiple seq_len

Description

Create multiple sequences of certain lengths at once.
It is a vectorized version of `lapply(x, seq_len)`.

Usage

```
seq_mlen(x)
```

Arguments

`x` an integer vector giving the lengths.

Value

A list with `length(x)` elements, where each element is the result of `seq_len`.

Examples

```
seq_mlen(c(10, 10, 3))
```

seq_names	<i>Generate Integer Sequence From a Range of Names</i>
-----------	--

Description

Generate integer sequence from a range of names.

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 w.
For a regular Fibonacci, inits = 0:1, which of course means a window size of w = 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 w.
The window size is equal to $w = \text{length}(\text{inits})$. |
| 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:
$f = \text{sum}$,
or (since window size is 2) $f = \backslash(x) \ x[2] + x[1]$ |

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
```

setapply

Apply Functions Over Mutable Atomic Matrix Margins By Reference

Description

The `setapply()` function applies a functions over the rows or columns of a [mutable_atomic](#) matrix, through [pass-by-reference semantics](#).

For every iteration, a copy of only a single row or column (depending on the margin) is made, the function is applied on the copy, and the original row/column is replaced by the modified copy through [pass-by-reference semantics](#).

The `setapply()` has about the same speed as [apply](#).
However, `setapply()` uses much less memory than [apply](#).

Usage

```
setapply(x, MARGIN, FUN)
```

Arguments

<code>x</code>	a mutable_atomic matrix. Arrays are not supported.
<code>MARGIN</code>	a single integer scalar, giving the subscript to apply the function over. 1 indicates rows, 2 indicates columns.
<code>FUN</code>	the function to be applied. In the case of (infix) operators, the function name must be backquoted or quoted.

Value

Returns: VOID. This function modifies the object by reference.
Do NOT use assignment like `x <- setapply(x, ...)`.
Since this function returns void, you'll just get NULL.

Examples

```
# re-order elements matrix by reference ====
x <- mutable_atomic(1:20, dim = c(5,4))
print(x)
setapply(x, 1, FUN = \(x)x[c(4,1,3,2)])
print(x)

# sort elements of matrix by reference ====
x <- mutable_atomic(20:1, dim = c(5,4))
print(x)
setapply(x, 2, FUN = sort)
print(x)
```

sub2ind	<i>Convert Subscripts to Coordinates, Coordinates to Flat Indices, and Vice-Versa</i>
---------	---

Description

These functions convert a list of integer subscripts to an integer matrix of coordinates, an integer matrix of coordinates to an integer vector of flat indices, and vice-versa. Inspired by the sub2ind function from 'MatLab'.

sub2coord() converts a list of integer subscripts to an integer matrix of coordinates.
 coord2ind() converts an integer matrix of coordinates to an integer vector of flat indices.
 ind2coord() converts an integer vector of flat indices to an integer matrix of coordinates.
 coord2sub() converts an integer matrix of coordinates to a list of integer subscripts.
 sub2ind() is a faster and more memory efficient version of coord2ind(sub2coord(sub, x.dims), x.dims) (especially for up to 5 dimensions).

Note that the coord2sub() function performs a very simple (one might even say naive) conversion.

All of these functions are written to be memory-efficient. The coord2ind() is thus the opposite of [arrayInd](#), and ind2coord is merely a convenient wrapper around [arrayInd](#).

Note that the equivalent to the sub2ind function from 'MatLab' is actually the coord2ind() function here.

Usage

```
sub2coord(sub, x.dim)

coord2sub(coord)

coord2ind(coord, x.dim, checks = TRUE)

ind2coord(ind, x.dim)

sub2ind(sub, x.dim, checks = TRUE)
```

Arguments

sub	<p>a list of integer subscripts.</p> <p>The first element of the list corresponds to the first dimension (rows), the second element to the second dimensions (columns), etc.</p> <p>The length of sub must be equal to the length of <code>x.dim</code>.</p> <p>One cannot give an empty subscript; instead fill in something like <code>seq_len(dim(x)[margin])</code>.</p> <p>NOTE: The <code>coord2sub()</code> function does not support duplicate subscripts.</p>
x.dim	an integer vector giving the dimensions of the array in question. I.e. <code>dim(x)</code> .
coord	<p>an integer matrix, giving the coordinate indices (subscripts) to convert.</p> <p>Each row is an index, and each column is the dimension.</p> <p>The first columns corresponds to the first dimension, the second column to the second dimensions, etc.</p> <p>The number of columns of coord must be equal to the length of <code>x.dim</code>.</p>
checks	<p>logical, indicating if arguments checks should be performed.</p> <p>Defaults to TRUE. Can be set to FALSE for minor speed improvements, but not recommended.</p>
ind	an integer vector, giving the flat position indices to convert.

Details

The S3 classes in 'R' use the standard Linear Algebraic convention, as in academic fields like Mathematics and Statistics, in the following sense:

- vectors are **column** vectors (i.e. vertically aligned vectors);
- index counting starts at 1;
- rows are the first dimension/subscript, columns are the second dimension/subscript, etc.

Thus, the orientation of flat indices in, for example, a 4 by 4 matrix, is as follows:

	[, 1]	[, 2]	[, 3]	[, 4]
[1,]	1	5	9	13
[2,]	2	6	10	14
[3,]	3	7	11	15
[4,]	4	8	12	16

The subscript `[1, 2]` refers to the first row and the second column. In a 4 by 4 matrix, subscript `[1, 2]` corresponds to flat index 5.

The functions described here thus follow also this convention.

Value

For `sub2coord()` and `ind2coord()`:

Returns an integer matrix of coordinates (with properties as described in argument `coord`).

For `coord2ind()`:

Returns an integer vector of flat indices (with properties as described in argument `ind`).

For coord2sub():

Returns a list of integer subscripts (with properties as described in argument sub)

Warning

These functions do not support duplicate indices.

For efficiency, they do not check for duplicate indices either.

Examples

```
x.dim <- c(10, 10, 3)
x.len <- prod(x.dim)
x <- array(1:x.len, x.dim)
sub <- list(c(4, 3), c(3, 2), c(2, 3))
coord <- sub2coord(sub, x.dim)
print(coord)
ind <- coord2ind(coord, x.dim)
print(ind)
all(x[ind] == c(x[c(4, 3), c(3, 2), c(2, 3)])) # TRUE
coord2 <- ind2coord(ind, x.dim)
print(coord)
all(coord == coord2) # TRUE
sub2 <- coord2sub(coord2)
sapply(1:3, \ (i) sub2[[i]] == sub[[i]]) |> all() # TRUE
```

Index

[.mutable_atomic
 (class_mutable_atomic), 21
[<-.mutable_atomic
 (class_mutable_atomic), 21

aaa0_squarebrackets, 2
aaa1_squarebrackets_immutable_classes,
 6
aaa2_squarebrackets_mutable_classes, 8
aaa3_squarebrackets_idx_args, 12
aaa4_squarebrackets_duplicates, 16
aaa5_squarebrackets_PassByReference,
 17
aaa6_squarebrackets_technical, 20
abind, 6, 30
apply, 54
array, 6, 22
arrayInd, 55
as.mutable_atomic
 (class_mutable_atomic), 21

baseenv, 18
bindingIsLocked, 17
BY, 26

c, 5, 30
class: array, 14
class: atomic, 13
class: data.frame-like, 13, 15
class: factor, 13, 15
class: list, 13
class: matrix, 13
class_mutable_atomic, 21
coercion_by_reference: depends, 10
coercion_by_reference: NO, 9, 10
coercion_by_reference: YES, 9
coercion_through_copy: depends, 7, 10
coercion_through_copy: NO, 7, 10
coercion_through_copy: YES, 6, 7, 9
collections, 4, 9
coord2ind, 5
coord2ind(sub2ind), 55
coord2sub(sub2ind), 55

dapply, 38, 45

data.frame, 3, 6
data.table, 3, 9, 34
drop, 16
dt, 23
dt_, 5
dt_aggregate(dt), 23
dt_setadd(dt), 23
dt_setcoe, 9, 33
dt_setcoe(dt), 23
dt_setrm(dt), 23

Extract, 3, 22

factor, 6
fastmap, 4, 9
first, 16
form, 15
format.mutable_atomic
 (class_mutable_atomic), 21
future_lapply, 38, 45

GENERIC METHODS, 5

HELPER FUNCTIONS, 5

idx_by, 5, 25
idx_ord_, 5
idx_ord_df(idx_ord_v), 27
idx_ord_m(idx_ord_v), 27
idx_ord_v, 27
immutable classes, 4
ind2coord(sub2ind), 55
indx_rm(indx_x), 28
indx_x, 28
is.mutable_atomic
 (class_mutable_atomic), 21

lapply, 38, 45, 46
last, 16
list, 5, 6, 30
lockBinding, 18, 35

match_all, 5, 29
matrix, 6
mutable classes, 4

- mutable object, [5](#)
- mutable_atomic, [4](#), [9](#), [18](#), [34](#), [54](#)
- mutable_atomic (class_mutable_atomic), [21](#)
- n, [5](#), [14](#), [15](#), [30](#)
- order, [27](#), [28](#)
- Pass By Reference, [5](#)
- pass-by-reference, [44](#)
- pass-by-reference semantics, [23](#), [54](#)
- print.mutable_atomic (class_mutable_atomic), [21](#)
- require unique names: NO, [21](#)
- require unique names: YES, [21](#)
- rm, [36](#)
- sample, [16](#)
- sb_a, [5](#)
- sb_a (sb_special), [47](#)
- sb_after, [5](#), [9](#)
- sb_after (sb_before), [30](#)
- sb_before, [5](#), [9](#), [30](#)
- sb_coe, [5](#), [21](#), [26](#), [33](#), [38](#)
- sb_currentBindings, [5](#), [18](#), [35](#)
- sb_mod, [5](#), [7](#), [9](#), [10](#), [21](#), [26](#), [37](#)
- sb_rec, [5](#), [21](#), [40](#)
- sb_rm, [5](#), [21](#), [26](#), [41](#)
- sb_set, [5](#), [7](#), [9](#), [10](#), [22](#), [23](#), [26](#), [44](#)
- sb_special, [47](#)
- sb_str, [5](#)
- sb_str (sb_special), [47](#)
- sb_x, [5](#), [15](#), [16](#), [21](#), [26](#), [49](#)
- seq_mlen, [5](#), [51](#)
- seq_names, [5](#), [52](#)
- seq_rec, [5](#), [52](#)
- setapply, [5](#), [54](#)
- setName, [22](#)
- sort, [28](#)
- SPECIALIZED FUNCTIONS, [5](#)
- squarebrackets (aaa0_squarebrackets), [2](#)
- squarebrackets-package (aaa0_squarebrackets), [2](#)
- squarebrackets_duplicates, [24](#), [38](#), [43](#), [45](#)
- squarebrackets_duplicates (aaa4_squarebrackets_duplicates), [16](#)
- squarebrackets_help (aaa0_squarebrackets), [2](#)
- squarebrackets_immutable_classes, [31](#), [34](#), [38](#), [42](#), [49](#)
- squarebrackets_immutable_classes (aaa1_squarebrackets_immutable_classes), [6](#)
- squarebrackets_indx_args, [24](#), [26](#), [28](#), [34](#), [38](#), [42](#), [45](#), [49](#)
- squarebrackets_indx_args (aaa3_squarebrackets_indx_args), [12](#)
- squarebrackets_mutable_classes, [17](#), [31](#), [34](#), [38](#), [42](#), [49](#)
- squarebrackets_mutable_classes (aaa2_squarebrackets_mutable_classes), [8](#)
- squarebrackets_PassByReference, [22](#), [35](#)
- squarebrackets_PassByReference (aaa5_squarebrackets_PassByReference), [17](#)
- squarebrackets_technical (aaa6_squarebrackets_technical), [20](#)
- sub2coord, [5](#)
- sub2coord (sub2ind), [55](#)
- sub2ind, [13](#), [55](#)
- supported mutable classes, [45](#)
- supported mutable object, [44](#)
- tibble, [3](#)
- tidytable, [3](#)
- vector, [6](#)