Package ‘squarebrackets’

December 1, 2024

Type Package

Title Subset Methods as Alternatives to the Square Brackets Operators for Programming

Version 0.0.0.9

Description Provides subset methods
(supporting both atomic and recursive S3 classes)
that may be more convenient alternatives to the `[` and `[<-` operators,
whilst maintaining similar performance.
Some nice properties of these methods include, but are not limited to, the following.
1) The `[` and `[<-` operators use different rule-sets for different data.frame-like types
(data.frames, data.tables, tibbles, tidytables, etc.).
The 'squarebrackets' methods use the same rule-sets for the different data.frame-like types.
2) Performing dimensional subset operations on an array using `[` and `[<-`,
requires a-priori knowledge on the number of dimensions the array has.
The 'squarebrackets' methods work on any arbitrary dimensions without requiring such prior knowledge.
3) When selecting names with the `[` and `[<-` operators,
only the first occurrence of the names are selected in case of duplicate names.
The 'squarebrackets' methods always perform on all names in case of duplicates,
not just the first.
4) The `[[` and `[[<-` operators
allow operating on a recursive subset of a nested list.
But these only operate on a single recursive subset,
and are not vectorized for multiple recursive subsets of a nested list at once.
'squarebrackets' provides a way to reshape a nested list
into a recursive matrix,
thereby allowing vectorized operations on recursive subsets of such a nested list.
5) The `[<-` operator only supports copy-on-modify semantics for most classes.
The 'squarebrackets' methods provides explicit pass-by-reference and pass-by-value semantics,
whilst still respecting things like binding-locks and mutability rules.
6) 'squarebrackets' supports index-less sub-set operations,
which is more memory efficient than sub-set operations using the `[` and `[<-` operators.

License MIT + file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Suggests rlang,

knitr,
rmarkdown,
tinytest,
tinycodet,
tidytable,
tibble,
ggplot2,
sf,
future.apply,
collections,
rrapply,
abind

Depends R (>= 4.2.0)

Imports Rcpp (>= 1.0.11),
collapse (>= 2.0.2),
data.table (>= 1.14.8),
stringi (>= 1.7.12)

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

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

Language en-gb

Contents

aaa00_squarebrackets_help . . . 3
aaa01_squarebrackets_supported_structures . . . 8
aaa02_squarebrackets_indx_fundamentals . . . 10
aaa03_squarebrackets_indx_args . . . 13
aaa04_squarebrackets_method_dispatch . . . 19
aaa05_squarebrackets_modify . . . 21
aaa06_squarebrackets_options . . . 24
aaa07_squarebrackets_PassByReference . . . 26
aaa08_squarebrackets_coercion . . . 29
aaa09_squarebrackets_inconveniences . . . 34
bind . . . 36
ci_flat . . . 39
class_mutable_atomic . . . 41
cp_seq . . . 44
currentBindings . . . 45
dt . . . 48
idx . . . 51
idx_by . . . 54
idx_ord_v . . . 55
idx_r . . . 57
indx_x . . . 58
lst . . . 58
match_all . . . 61
ma_setv . . . 62
n . . . 63
ndims . . . 64

sb2_rec . . . 65
sb_mod . . . 67
sb_rm . . . 72
sb_set . . . 75
sb_setRename . . . 78
sb_x . . . 80
setapply . . . 83
slice . . . 84
sub2ind . . . 85
tci_bool . . . 87

Index **90**

`aaa00_squarebrackets_help` *squarebrackets: Subset Methods as Alternatives to the Square Brackets Operators for Programming*

Description

squarebrackets: Subset Methods as Alternatives to the Square Brackets Operators for Programming

Provides subset methods (supporting both atomic and recursive S3 classes) that may be more convenient alternatives to the `[` and `[<-` operators, whilst maintaining similar performance.
Some nice properties of these methods include, but are not limited to, the following.

1. The `[` and `[<-` operators use different rule-sets for different data.frame-like types (data.frames, data.tables, tibbles, tidytables, etc.).
   The 'squarebrackets' methods use the same rule-sets for the different data.frame-like types.
2. Performing dimensional subset operations on an array using `[` and `[<-`, requires a-priori knowledge on the number of dimensions the array has.
   The 'squarebrackets' methods work on any arbitrary dimensions without requiring such prior knowledge.
3. When selecting names with the `[` and `[<-` operators, only the first occurrence of the names are selected in case of duplicate names.
   The 'squarebrackets' methods always perform on all names in case of duplicates, not just the first.
4. The `[[` and `[[<-` operators allow operating on a recursive subset of a nested list.
   But these only operate on a single recursive subset, and are not vectorized for multiple recursive subsets of a nested list at once.
   'squarebrackets' provides a way to reshape a nested list into a recursive matrix, thereby allowing vectorized operations on recursive subsets of such a nested list.
5. The `[<-` operator only supports copy-on-modify semantics for most classes.
   The 'squarebrackets' methods provides explicit pass-by-reference and pass-by-value semantics, whilst still respecting things like binding-locks and mutability rules.
6. 'squarebrackets' supports index-less sub-set operations, which is more memory efficient than sub-set operations using the `[` and `[<-` operators.

Goal

Among programming languages, 'R' has perhaps one of the most flexible and comprehensive sub-setting functionality, provided by the square brackets operators (`[`, `[<-`).
But in some situations the square brackets operators are occasionally less than optimally convenient

The Goal of the 'squarebrackets' package is not to replace the square-brackets operators, but to provide **alternative** sub-setting methods and functions, to be used in situations where the square bracket operators are inconvenient.

Supported Structures

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

'squarebrackets' supports the following immutable structures:

- basic `atomic` classes
  (atomic vectors, matrices, and arrays).
- factor.
- basic list classes
  (recursive vectors, matrices, and arrays).
- data.frame
  (including the classes `tibble`, `sf-data.frame` and `sf-tibble`).

'squarebrackets' supports the following mutable structures:

- mutable_atomic
  (`mutable_atomic` vectors, matrices, and arrays);
- data.table
  (including the classes `tidytable`, `sf-data.table`, and `sf-tidytable`).

See squarebrackets_supported_structures for more details.

Sub-set Operation Methods & Binding Implementations

The main focus of this package is on its generic methods and dimensional binding implementations.

Generic methods for atomic objects start with `sb_`.
Generic methods for recursive objects (list, data.frame, etc.) start with `sb2_`.
There is also the somewhat separate idx method, which works on both recursive and non-recursive

objects.
The binding implementations for dimensional objects start with `bind_`.
And finally there are the `slice_` methods, which (currently) only work on (mutable) atomic vectors.

ACCESS SUBSETS

Methods to access subsets (i.e. extract selection, or extract all except selection):

- sb_x, sb2_x: extract, exchange, or duplicate subsets.
- sb_rm, sb2_rm: un-select/remove subsets.
- sb2_rec: access recursive subsets of lists.
- slice_x, slice_rm: efficiently extract or un-select/remove subset from a (long) vector.

MODIFY SUBSETS

Methods to modify subsets:

- idx: translate given indices/subscripts, for the purpose of copy-on-modify substitution.
- sb2_recin: replace, transform, remove, or add recursive subsets to a list, through R's default Copy-On-Modify semantics.
- sb_mod, sb2_mod: return the object with modified (transformed or replaced) subsets.
- Methods to rename a mutable object using pass-by-reference semantics.
- sb_set, sb2_set: modify (transform or replace) subsets of a mutable object using pass-by-reference semantics.
- slice_set: efficiently modify a (long) vector subset using pass-by-reference semantics.

EXTEND BEYOND

Methods and binding implementations, to extend or re-arrange an object beyond its current size:

- bind_: implementations for binding dimensional objects.
- sb_x, sb2_x: extract, exchange, or duplicate subsets.
- sb2_recin: replace, transform, remove, or add recursive subsets to a list, through R's default Copy-On-Modify semantics.

See squarebrackets_method_dispatch for more information on how 'squarebrackets' uses its S3 Method dispatch.

Functions

SPECIALIZED FUNCTIONS
Additional specialized sub-setting functions are provided:

- lst_untree: unnest tree-like nested list into a recursive matrix, to speed-up vectorized sub-setting on recursive subsets of the list.
- The dt_-functions to programmatically perform `data.table`-specific `[`-operations, with the security measures provided by the 'squarebrackets' package.
- setapply: apply functions over mutable matrix margins using pass-by-reference semantics.
- ma_setv: Find & Replace values in mutable_atomic objects using pass-by-reference semantics.
  This is considerably faster and more memory efficient than using sb_set for this.

HELPER FUNCTIONS
A couple of convenience functions, and helper functions for creating ranges, sequences, and indices (often needed in sub-setting) are provided:

- currentBindings: list or lock all currently existing bindings that share the share the same address as the input variable.
- n: Nested version of c, and short-hand for list.
- ndims: Get the number of dimensions of an object.
- sub2coord, coord2ind: Convert subscripts (array indices) to coordinates, coordinates to flat indices, and vice-versa.
- match_all: Find all matches, of one vector in another, taking into account the order and any duplicate values of both vectors.
- Computing indices:
  idx_r to compute an integer index range.
  idx_by to compute grouped indices.
  idx_ord_-functions to compute ordered indices.

DEVELOPER FUNCTIONS

And finally some developer functions for constructing indices.
These are also used internally by 'squarebrackets', and package authors can use these to create additional sb_/sb2_ S3 methods, or even entirely new subset-related functions.

- tci_ functions, for type-casting indices.
- ci_ functions, for constructing indices.
- indx_x and indx_rm, for testing methods.

Properties Details

The alternative sub-setting methods and functions provided by 'squarebrackets' have the following properties:

- **Programmatically friendly**:
  - Unlike base `[`, it's not required to know the number of dimensions of an array a-priori, to perform subset-operations on an array.

  - 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 methods 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 (partial) 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 **(partial) copy**.
- **Careful handling of names**:
  - Sub-setting an object by index names returns ALL matches with the given names, not just the first.
  - Data.frame-like objects (see supported classes below) are forced to have unique column names.
  - Sub-setting arrays using `x[indx1, indx2, etc.]` will drop `names(x)`.
    The methods from 'squarebrackets' will not drop `names(x)`.
- **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.

Author(s)

Author, 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/`

aaa01_squarebrackets_supported_structures
Supported Structures

Description

'squarebrackets' only supports the most common S3 objects, and only those that primarily use square brackets for sub-set operations (hence the name of the package).

One can generally divide the structures supported by 'squarebrackets' along 3 key properties:

- atomic vs recursive:
  Types `logical`, `integer`, `double`, `complex`, `character`, and `raw` are `atomic`.
  Lists and data.frames are `recursive`.
- dimensionality:
  Whether an object is a vector, matrix, array, or data.frame.
- mutability:
  Base R's S3 classes (except Environments) are generally immutable:
  Modifying the object will create a copy (called 'copy-on-modify').
  'squarebrackets also supports `data.tables` and mutable_atomic objects, which are mutable:
  If desired, one can modify them without copy using pass-by-reference semantics.

Supported Structures

'squarebrackets' supports the following immutable structures:

- basic `atomic` classes
  (atomic vectors, matrices, and arrays).
- factor.
- basic list classes
  (recursive vectors, matrices, and arrays).
- data.frame
  (including the classes `tibble`, `sf-data.frame` and `sf-tibble`).

'squarebrackets' supports the following mutable structures:

- mutable_atomic
  (`mutable_atomic` vectors, matrices, and arrays);
- data.table
  (including the classes `tidytable`, `sf-data.table`, and `sf-tidytable`).

Details

Atomic vs Recursive

The `sb_` methods provided by 'squarebrackets' work on **atomic** (see is.atomic) objects.
The `sb2_` methods provided by 'squarebrackets' work on **recursive** (see is.recursive) objects.
See squarebrackets_method_dispatch for more details on the method dispatch used by 'squarebrackets'.

Dimensionality

'squarebrackets' supports dimensionless or vector objects (i.e. ndims == `0L`).
squarebrackets' supports arrays (see is.array and is.matrix); note that a matrix is simply an array with 2 dimensions.
squarebrackets' also supports data.frame-like objects (see is.data.frame).
Specifically, squarebrackets' supports a wide variety of data.frame classes:
`data.frame`, `data.table`, `tibble`, `tidytable`;
'squarebrackets' also supports their 'sf'-package compatible counter-parts:
`sf-data.frame`, `sf-data.table`, `sf-tibble`, `sf-tidytable`.

Dimensionless vectors and dimensional arrays are supported in both their atomic and recursive forms.
Data.frame-like objects, in contrast, **only** exist in the **recursive** form (and, as stated, are supported by 'squarebrackets').
Recursive vectors, recursive matrices, and recursive arrays, are collectively referred to as "lists" in the 'squarebrackets' documentation.

Note that the dimensionality of data.frame-like objects is not the same as the dimensionality of (recursive) arrays/matrices.
For any array/matrix `x`, it holds that `length(x) == prod(dim(x))`.
But for any data.frame `x`, it is the case that `length(x) == ncol(x)`.

Mutable vs Immutable

Most S3 objects in base 'R' are immutable:
They have no explicit pass-by-reference semantics.
Environments do have pass-by-reference semantics, but they are not supported by 'squarebrackets'.

'squarebrackets' supports the mutable `data.table` class (and thus also `tidytable`, which inherits from `data.table`).

'squarebrackets' also includes a new class of mutable objects: mutable_atomic objects.
`mutable_atomic` objects are the same as atomic objects, except they are mutable (hence the name).

The supported immutable structures are:
Atomic and recursive vectors/matrices/arrays, data.frames, and tibbles.

All the functions in the 'squarebrackets' package with the word "set" in their name perform pass-by-reference modification, and thus only work on mutable structures.
All other functions work the same way for both mutable and immutable structures.

Derived Atomic Vector

A special class of objects are the Derived Atomic Vector structures:
structures that are derived from atomic objects, but behave differently.
For example:
Factors, datetime, POSIXct and so on are derived from atomic vectors.
But they have attributes and special methods that make them behave differently.

'squarebrackets' treats derived atomic classes as regular atomic vectors.
There are highly specialized packages to handle objects derived from atomic objects.
For example, the 'anytime' package to handle date-time objects.

'squarebrackets does provide some more explicit support for factors.

Not Supported S3 structures
Key-Values storage S3 structures, such as environments, are not supported by 'squarebrackets'.

`aaa02_squarebrackets_indx_fundamentals`
Indexing Fundamentals

Description

This help page explains the fundamentals regarding how 'squarebrackets' treats indexing.

Indexing Types

Base 'R' supports indexing through `logical`, `integer`, and `character` vectors.
'squarebrackets' supports these also (albeit with some improvements), but also supports some additional methods of indexing.

Whole numbers
Whole numbers are the most basic form on index selection.
All forms of indexing in 'squarebrackets' are internally translated to integer (or double if `> (2^31 - 1)`) indexing first, ensuring consistency.
Indexing through integer/numeric indices in 'squarebrackets' works the same as in base 'R', except that negative values are not allowed.

Logical
Selecting indices with a logical vector in 'squarebrackets' works the same as in base 'R', except that recycling is not allowed.
Thus the logical vector must be of the correct length (i.e. `length(x)` or `dim(x)[L]`, depending on the situation).

Characters

When selecting indices using a character vector, base 'R' only selects the first matches in the names. 'squarebrackets', however, selects all matches.
Character indices are internally translated to integer indices using match_all.

Imaginary Numbers
A complex vector `y` is structured as
`y = a + b * i`
where `Re(y)` returns `a`, and `Im(y)` returns `b`.
squarebrackets' includes support for indexing through imaginary numbers (`Im(y)`) of complex vectors.
Indexing with imaginary numbers is a generalization of indexing with regular integers.

It works as follows:
Imaginary numbers that are positive integers, like `1:10 * 1i`, work the same as regular integers.
Imaginary numbers that are negative integers, like `1:10 * -1i`, index by counting backwards (i.e. from the end), where the integer indices are computed as `n + Im(y) + 1L`.
Here `n` is the maximum possible integer (i.e. `length(x)`, or `dim(x)[L]`, depending on the situation), and `Im(y)` is negative.
Note that **only** the `Imaginary` part of a complex vector is used (`Im(y)`);
the `Real` part (`Re(y)`) is **ignored**.

See the results of the following code as an example:

```
x <- 1:30 # vector of 30 elements

sb_x(x, 1:10 * 1i) # extract first 10 elements
#>  [1]  1  2  3  4  5  6  7  8  9 10

sb_x(x, 1:10 * -1i) # extract last 10 elements
#>  [1] 30 29 28 27 26 25 24 23 22 21

sb_x(x, 10:1 * -1i) # last 10 elements, in tail()-like order
#>  [1] 21 22 23 24 25 26 27 28 29 30
```

Thus complex vectors allow the user to choose between counting from the beginning, like regular integers, or counting from the end.

Subscripts

One can distinguish between flat indices, array subscripts, and data.frame subscripts.

Flat indices, also called linear indices, specifies the index of a vector, ignoring dimensions (if any are present).
So in an expression like `x[i]`, where `i` is a vector, `i` specifies flat indices.

Matrices and arrays also have array subscripts.
Array subscripts works by specifying multiple indexing vectors, which can be of different sizes, where each vector specifies positions in a specific dimension.
Given, for example, a 3-dimensional array, the subscript `[1:10, 2:5, 3:9]` refers to rows 1 to 10,

columns 2 to 5, and layers 3 to 9.
The base S3 vector 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-rows-by-5-columns matrix, is as follows:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

In a 4 by 4 matrix, subscript `[1,2]` corresponds to flat index `5`.
All array subscripts in 'squarebrackets' also follow this convention.

Data.frame-like objects use data.frame subscripts.
At first glance this may seem the same as the array subscripts of matrices, but they are not:
The column indices of a data.frame-like object is equal to its flat indices.
I.e. for a data.frame, `x[i]` is essentially the same as `x[, i]`, safe for some attribute handling.
To avoid confusion, 'squarebrackets' does not have an argument for flat indices in its data.frame methods.

Flat indices (or just "indices" for non-dimensional objects) exist for all objects (in data.frame-like objects, flat indices are actually equal to column indices).
Thus flat indices are the "default" indices, and are usually just referred to as simply "indices".

Indexing in Recursive Subsets

One of the differences between atomic and recursive objects, is that recursive objects support recursive subsets, while atomic objects do not.

Bear in mind that every element in a recursive object is a reference to another object.
Consider the following list `x`:

```
x <- list(A = letters, B = 1:10, C = list(A = 11:20, B = LETTERS))
```

Regular subsets, AKA surface-level subset operations (`[`, `[<-` in base 'R'), operate on the recursive object itself.
I.e. sb2_x`(x, 1)`, or equivalently `x[1]`, returns the list `list(A = letters)`.
Recursive subset operations (`[[`, `[[<-` in base 'R'), on the other hand, operate on an object a subset of the recursive object references to.
I.e. sb2_rec`(x, 1)`, or equivalently `x[[1]]`, returns the **character vector** `letters`.

Recursive objects can refer to other recursive objects, which can themselves refer to recursive objects, and so on.
Recursive subsets can go however deep you want.
So, for example, to extract the character vector `LETTERS` from the aforementioned list `x`, one would need to do:
sb2_rec`(x, c("C","B"))`, or equivalently, `x[["C"]][["B"]]`.
You can also do this using integers of course: sb2_rec`(x, c(3, 2))`.

Note that recursive subset operations using sb2_rec/sb2_recin only support positive integer vectors and character vectors;
imaginary numbers (using complex vectors) and logical vectors are not supported.
Moreover, since a recursive subset operation only operates on a single element, specifying the index with a character vector only selects the first matching element (just like base 'R'), not all matches.

Regarding Performance

Integer indices and logical indices are the fastest.
Indexing through names (i.e. character vectors) is the slowest.
Thus if performance is important, use integer or logical indices.

`aaa03_squarebrackets_indx_args` *Index Arguments in the Generic Sub-setting Methods*

Description

There are several 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.
- `sub, dims`: to specify indices of arbitrary dimensions in arrays (including matrices, which inherit from arrays).
- `margin, slice`: to specify indices of one particular dimension.
- `row, col`: to specify rows and/or columns in specifically in data.frame-like objects.
- `filter, vars`: to specify rows and/or columns specifically in data.frame-like objects.

Thus there are essentially 3 APIs: one for vectors, one for arrays and matrices, and one for data.frame-like objects.

For the fundamentals of indexing in 'squarebrackets', see squarebrackets_indx_fundamentals.
In this help page `x` refers to the object on which subset operations are performed.

Argument i

class: atomic vector
class: derived atomic vector
class: recursive vector

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

- `NULL`, corresponds to missing argument.
- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a numeric vector of **strictly positive whole numbers** with indices.
- a **complex** vector, as explained in squarebrackets_indx_fundamentals.
- a **logical vector**, 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 recursive objects, `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]   # if `x` is atomic
sb2_x(x, i = i) # ==> x[i]  # if `x` is recursive
```

If `i` is a function, it corresponds to the following:

```
sb_x(x, i = i) # ==> x[i(x)] # if `x` is atomic
sb2_x(x, i = i) # ==> x[lapply(x, i)] # if `x` is recursive
```

Argument Pair sub, dims

class: atomic matrix
class: recursive matrix
class: atomic array
class: recursive array
The `sub, dims` argument pair is inspired by the `abind::`asub function from the 'abind' package (see reference below).
`dims` must be an integer vector, giving the dimensions for which to specify the subscripts. (i.e. `dims` specifies the "non-missing" margins).

`sub` must be either of the following:

- a list of length `length(dims)`.

- a list of length 1;
  in this case `sub` will be recycled to `length(dims)`.
- an atomic vector;
  this is functionally equivalent to specifying `sub` as a list of length 1.

Each element of `sub` when `sub` is a list, or `sub` itself when `sub` is an atomic vector, can be any of the following:

- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a numeric vector of **strictly positive whole numbers** with indices of the specified dimension to select for the operation.
- a **complex** vector, as explained in squarebrackets_indx_fundamentals.
- a **logical** vector of the same length as the corresponding dimension size, giving the indices of the specified dimension to select for the operation.
- a **character** vector giving the `dimnames` to select.
  If a dimension has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

Note also the following:

- As stated, `dims` specifies which index margins are non-missing.
  If `dims` is of length `0`, it is taken as "all index margins are missing".
- The default value for `dims` is `1:ndims(x)`.

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

Here are some examples for clarity, using an array `x` of 3 dimensions:

- `sb_x(x, n(1:10, 1:5), c(1, 3))`
  extracts the first 10 rows, all columns, and the first 5 layers, of array `x`.
  The equivalence in base 'R' is:
  `x[1:10, , 1:5, drop = FALSE]`.
- `sb_x(x, n(1:10), c(1, 3))`, or equivalently `sb_x(x, 1:10, c(1, 3))`,
  extracts the first 10 rows, all columns, and the first 10 layers, of array `x`.
  The equivalence in base 'R' is:
  `x[1:10, , 1:10, drop = FALSE]`.
- `sb_x(x, n(1:10))`, or equivalently `sb_x(x, 1:10)`,
  extracts the first 10 rows, columns, and layers of array `x`.
  The equivalence in base 'R' is:
  `x[1:10, 1:10, 1:10, drop = FALSE]`.

I.e.:

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

sb_x(x, n(1:10), c(1, 3))      # ==> x[1:10, , 1:10, drop = FALSE]
sb2_x(x, 1:10, c(1, 3))        # ==> x[1:10, , 1:10, drop = FALSE]

sb2_x(x, n(1:10))              # ==> x[1:10, 1:10, 1:10, drop = FALSE]
sb_x(x, 1:10)                  # ==> x[1:10, 1:10, 1:10, drop = FALSE]
```

Note that specifying a list of length 1 for `sub` (like `sub = n(1:10)`) is equivalent to specifying an atomic vector for `sub` (like `sub = 1:10`).

For a brief explanation of the relationship between flat indices (`i`), and dimensional subscripts (`sub`, `dims`), see squarebrackets_indx_fundamentals.

Argument Pair margin, slice

class: atomic matrix
class: recursive matrix
class: atomic array
class: recursive array
class: data.frame-like

Relevant only for the idx method.
The `margin` argument specifies the dimension on which argument `slice` is used.
I.e. when `margin = 1`, `slice` selects rows;
when `margin = 2`, `slice` selects columns;
etc.

The `slice` argument can be any of the following:

- a numeric vector of **strictly positive whole numbers** with dimension indices to select for the operation.
- a **complex** vector, as explained in squarebrackets_indx_fundamentals.
- a **logical** vector of the same length as the corresponding dimension size, giving the dimension indices to select for the operation.
- a **character** vector of index names.
  If a dimension has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

One could also give a vector of length `0` for `slice`;
Argument `slice` is only used in the idx method , and the result of idx are meant to be used inside the regular `[` and `[<-` operators.
Thus the effect of a zero-length index specification depends on the rule-set of `[.class(x)` and `[<-.class(x)`.

Arguments row, col

class: data.frame-like

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

- `NULL` (default), corresponds to a missing argument.
- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a numeric vector of **strictly positive whole numbers** with indices of the specified dimension to select for the operation.
- a **complex** vector, as explained in squarebrackets_indx_fundamentals.
- a **logical** vector of the same length as the corresponding dimension size, giving the indices of the specified dimension to select for the operation.
- a **character** vector giving the `dimnames` to select.
  If a dimension has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

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

```
sb2_x(x, row, col) # ==> x[row, col, drop = FALSE]
```

Arguments filter, vars

class: data.frame-like

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

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

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

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

Argument inv

all classes

Relevant for the sb_mod/sb2_mod, sb_set/sb2_set, and idx methods.
By default, `inv = FALSE`, which translates the indices like normally.
When `inv = TRUE`, the inverse of the indices is taken.
Consider, for example, an atomic matrix `x`;
using `sb_mod(x, 1:2, 2L, tf = tf)` corresponds to something like the following:

```
x[, 1:2] <- tf(x[, 1:2])
x
```

and using `sb_mod(x, col = 1:2, inv = TRUE, tf = tf)` corresponds to something like the following:

```
x[, -1:-2] <- tf(x[, -1:-2])
x
```

NOTE
The order in which the user gives indices when `inv = TRUE` generally does not matter.
The order of the indices as they appear in the original object `x` is maintained, just like in base 'R'.
Therefore, when replacing multiple values where the order of the replacement matters, it is better to keep `inv = FALSE`, which is the default.
For replacement with a single value or with a transformation function, `inv = TRUE` can be used without considering the ordering.

All NULL indices

`NULL` in the indexing arguments corresponds to a missing argument.
Thus, for **both** sb_x and sb_rm, using `NULL` for all indexing arguments corresponds to something like the following:

```
x[]
```

Similarly, for sb_mod and sb_set, using `NULL` corresponds to something like the following:

```
x[] <- rp # for replacement
x[] <- tf(x) # for transformation
```

The above is true **even if** `inv = TRUE` and/or `red = TRUE`.

Out-of-Bounds Integers, Non-Existing Names, and NAs

- Integer indices that are out of bounds (including `NaN` and `NA_integer_`) always give an error.
- Specifying non-existing names (including `NA_character_`) as indices is considered a form of zero-length indexing.
- Logical indices are translated internally to integers using which, and so `NA`s are ignored.

Disallowed Combinations of Index Arguments

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

One cannot specify `row` and `filter` simultaneously; it's either one or the other.
One cannot specify `col` and `vars` simultaneously; it's either one or the other.
One cannot specify the `sub, dims` pair and `slice, margin` pair simultaneously; it's either one pair or the other pair.
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, lyr, sub, 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]
```

References

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

`aaa04_squarebrackets_method_dispatch`
Method Dispatch of 'squarebrackets'

Description

This help page gives some additional details regarding the S3 method dispatch used in 'squarebrackets'.

Atomic vs Recursive

Atomic and recursive objects are quite different from each other in some ways:

- **homo- or heterogeneous**: an atomic object can only have values of one data type.
  recursive objects can hold values of any combination of data types.
- **nesting**: Recursive objects can be nested, while atomic objects cannot be nested.
- **copy and coercion effect**: One can coerce or copy a subset of a recursive object, without copying the rest of the object.
  For atomic objects, however, a coercion or copy operation coerces or copies the entire vector (ignoring attributes).

- **vectorization**: most vectorized operations generally work on atomic objects, whereas recursive objects often require loops or apply-like functions.

- **recursive subsets**: Recursive objects distinguish between "regular" subset operations (in base R using `[`, `[<-`), and recursive subset operations (in base R using `[[`, `[[<-`).
  See for example the sb2_rec method, or the `red = TRUE` argument in the sb2_x and sb2_rm methods.
  For atomic objects, these 2 have no meaningful difference (safe for perhaps some minor attribute handling).

- **views**: For recursive objects, one can create a view of a recursive subset.
  Subset views do not exist for atomic objects.

The main S3 methods that perform subset operation on an object, come in the atomic (`sb_`) and recursive (`sb2_`) form.
The idx method operates on the indices of an object, but does not operate on the object itself, and so has no distinction between the atomic and recursive form.

The split between the atomic and recursive forms of the method dispatches is done for several reasons:

- There are too many nuances to keep track of for the user between atomic and recursive objects. By splitting the methods into atomic and recursive objects, the user only has to choose `sb_` or `sb2_`, and 'squarebrackets' can handle most of the rest for the user.
- By giving atomic and recursive separate methods, it becomes syntactically clear what the consequences are for a subset-operation:
  will the entire object be coerced or copied? will a transformation function go through `lapply`? is an operation only affecting shallow subsets? etc.
- Some S3 classes, like the `array` and `matrix` classes, are available in both atomic and recursive forms.
  But the S3 method dispatch does not distinguish between atomic and recursive objects, despite the aforementioned differences between the 2.
  So 'squarebrackets' uses a separate method dispatch for the atomic and recursive form.
- Package authors can create separate sub-set operation methods for atomic and recursive objects using 'squarebrackets'.

Manual Dispatch

The 'squarebrackets' package intentionally exports each function in its S3 method dispatch system.
This is handy for programming purposes.
For example: one can explicitly alias a specific dispatch of a method, if one so desires.
For example like so:

```
array_x <- function(x, ...) {

   if(is.atomic(x)) {
     sb_x.array(x, ...)
```

```
    }
    else if(is.recursive(x)) {
      sb2_x.array(x, ...)
    }

}
```

Under certain circumstances, this might help your code to be more clear.

Ellipsis

Due to how the S3 method dispatch system works in 'R', all generic methods have the ellipsis argument (`...`).
For the user's safety, 'squarebrackets' does check that the user doesn't accidentally add arguments that make no sense for that method (like specifying the `inv` argument when calling sb_x).

`aaa05_squarebrackets_modify` *Regarding Modification*

Description

This help page describes the main modification semantics available in 'squarebrackets'.

Base R's default modification

For most average users, R's default copy-on-modify semantics are fine.
The benefits of the indexing arguments from 'squarebrackets' can be combined the `[<-` operator, through the idx method.
The result of the `idx()` method can be used inside the regular square-brackets operators.
For example like so:

```
x <- array(...)
my_indices <- idx(x, sub, dims)
x[my_indices] <- value

y <- data.frame(...)
rows <- idx(y, 1:10, 1, inv = TRUE)
cols <- idx(y, c("a", "b"), 2)
y[rows, cols] <- value
```

thus allowing the user to benefit from the convenient index translations from 'squarebrackets', whilst still using R's default copy-on-modification semantics (instead of the semantics provided by 'squarebrackets').

Explicit Copy

'squarebrackets' provides the sb_mod/sb2_mod method to modify through copy.
This method always copies the modification.
For recursive objects, sb2_mod returns the original object, where only the modified subsets are copied, thus preventing unnecessary usage of memory.

Pass-by-Reference

'squarebrackets' provides the sb_set/sb2_set and slice_set methods to modify by reference, meaning no copy is made at all.
Pass-by-Reference is fastest and the most memory efficient.
But it is also more involved than the other modification forms, and requires more thought.
See squarebrackets_PassByReference for more information.

Replacement and Transformation in Atomic Objects

The `rp` argument is used to replace the values at the specified indices with the values specified in `rp`.
Using the `rp` argument in the modification methods, corresponds to something like the following:

```
x[...] <- rp
```

The `tf` argument is used to transform the values at the specified indices through transformation function `tf`. Using the `tf` argument corresponds to something like the following:

```
x[...] <- tf(x[...])
```

where `tf` is a function that **returns** an object of appropriate type and size (so `tf` should not be a pass-by-reference function).

Replacement and Transformation in Lists

The `rp` and `tf` arguments work mostly in the same way for recursive objects.
But there are some slight differences.

Argument `rp`
'squarebrackets' demands that `rp` is always provided as a list in the S3 methods for recursive vectors, matrices, and arrays (i.e. lists).
This is to prevent ambiguity with respect to how the replacement is recycled or distributed over the specified indices
(See `Footnote 1` below).

Argument `tf`
Most functions in (base) 'R' are vectorized for atomic objects, but not for lists
(see `Footnote 2` below).
'squarebrackets' will therefore apply transformation function `tf` via `lapply`, like so:

```
x[...] <- lapply(x[...], tf)
```

In the methods for recursive objects, the `tf` argument is accompanied by the `.lapply` argument.
By default, `.lapply = lapply`.
The user may supply a custom `lapply()`-like function in this argument to use instead.
For example, the 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.

Replacement and Transformation in data.frame-like Objects

Replacement and transformations in data.frame-like objects are a bit more flexible than in Lists.

`rp` is not always demanded to be a list for data.frame-like objects, only when appropriate (for example, when replacing multiple columns, or when the column itself is a list.)

Bear in mind that every column in a data.frame is like an element in a list;
so `.lapply` is used for transformations across multiple columns.

Recycling and Coercion

Recycling is not allowed in the modification methods.
So, for example, `length(rp)` must be equal to the length of the selected subset, or equal to `1`.
The user should also take into account the auto-coercion rules of the object's class.
See squarebrackets_coercion for details.

Footnotes

Footnote 1

Consider the following replacement in base 'R':

```
x <-list(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
x[1:2] <- 2:1
```

What will happen?
Will the `x[1]` be `list(1:2)` and `x[2]` also be `list(1:2)`?
Or will `x[1]` be `list(2)` and `x[2]` be `list(1)`?
It turns out the latter will happen; but this is somewhat ambiguous from the code.
To prevent such ambiguity in your code, 'squarebrackets' demands that `rp` is always provided as a list.

Footnote 2

Most functions in (base) 'R' are vectorized for atomic objects, but not for lists.
One of the reasons is the following:
In an atomic vector `x` of some type `t`, every single element of `x` is a scalar of type `t`.
However, every element of some list `x` can be virtually anything:
an atomic object, another list, an unevaluated expression, even dark magic like `quote(expr =)`.
It is difficult to make a vectorized function for an object with so many unknowns.
Therefore, in the vast majority of the cases, one needs to loop through the list elements.

`aaa06_squarebrackets_options`

squarebrackets Options

Description

This help page explains the various global options that can be set for the 'squarebrackets' package, and how it affects the functionality.

Check Duplicates

argument: chkdup
option: squarebrackets.chkdup
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`).

Setting chkdup = TRUE 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.

Since checking for duplicates can be expensive, it is set to FALSE by default.
The default can be changed in the squarebrackets.chkdup option.

Sticky

argument: sticky
option: squarebrackets.sticky
The slice_x and slice_rm methods can already handle names, attributes specific to the mutable_atomic class, and attributes specific to the factor class.

When sticky = FALSE, which is arguably the safest setting, the slice_x and slice_rm methods will drop all **other** attributes.

By setting sticky = TRUE, all attributes except comment and tsp will be preserved;
name-related attributes are separate and are handled by the use.names argument.
The key advantage for this, is that classes that use static attributes (i.e. classes that use attributes that do not change when sub-setting), are automatically supported if sticky = TRUE, and no separate methods have to written for slice_x and slice_rm.
Attributes specific to classes like difftime, Date, POSIXct, roman, hexmode, octmode, and more, use static attributes.

Instead of setting sticky = TRUE or sticky = FALSE, one can also specify all classes that use static attributes that you'll be using in the current R session.
In fact, when 'squarebrackets' is loaded (**loaded**, attaching is not necessary), the squarebrackets.sticky option is set as follows:

```
squarebrackets.sticky = c(
   "difftime", "Date", "POSIXct", "roman", "hexmode", "octmode"
)
```

So in the above default setting, sticky = TRUE for
"difftime", "Date", "POSIXct", "roman", "hexmode", "octmode".
Also in the above default setting, sticky = FALSE for other classes.
Note, again, that mutable_atomic and and factor are already handled by slice_x and slice_rm, and their handling is **not** affected by the sticky argument/option.

The reason the slice_x and slice_rm need the sticky option, is because of the following.
Unlike most sb_/sb2_ methods, the slice_x and slice_rm methods are not wrappers around the [and [<- operators.
Therefore, most [- S3 methods for highly specialized classes are not readily available for the slice_x and slice_rm methods.
And therefore, important class-specific attributes are not automatically preserved.
The sticky option is a convenient way to support a large number of classes, without having to write

specific methods for them.

For specialized class that use attributes that **do** change when sub-setting, separate slice_x and slice_rm methods need to be written.
Package authors are welcome to create method dispatches for their own classes for these methods.

As a final note, the name "sticky" is inspired by `sticky::`sticky.

`aaa07_squarebrackets_PassByReference`
Regarding Modification By Reference

Description

This help page describes how modification using "pass-by-reference" semantics is handled by the 'squarebrackets' package.
This help page does not explain all the basics of pass-by-reference semantics, as this is treated as prior knowledge.
All functions/methods in the 'squarebrackets' package with the word "set" in the name use pass-by-reference semantics.

Advantages and Disadvantages

The main advantage of pass-by-reference is that much less memory is required to modify objects, and modification is also generally faster.
But it does have several disadvantages.

First, the coercion rules are slightly different: see squarebrackets_coercion.

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 <- mutable_atomic(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,

```
x <- mutable_atomic(1:16)
y <- x
lockBinding("y", environment())
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 Classes

With the exception of environments, most of base R's S3 classes are treated as immutable:
Modifying an object in 'R' will make a copy of the object, something called 'copy-on-modify' semantics.

A prominent mutable S3 class is the `data.table` class, which is a mutable data.frame class, and supported by 'squarebrackets'.
Similarly, 'squarebrackets' adds a class for mutable atomic objects:
mutable_atomic.

Material vs Immaterial objects

Most objects in 'R' are material objects:
the values an object contains are actually stored in memory.
For example, given `x <- rnorm(1e6)`, `x` is a material object:
1 million values (decimal numbers, in this case) are actually stored in memory.

In contrast, ActiveBindings are immaterial:
They are objects that, when accessed, call a function to generate values on the fly, rather than actually storing values.

Since immaterial objects do not actually store the values in memory, the values obviously also cannot be changed in memory.
Therefore, Pass-by-Reference semantics don't work on immaterial objects.

ALTREP

The mutable_atomic constructors (i.e. mutable_atomic, as.mutable_atomic, etc.) will automatically materialize ALTREP objects, to ensure consistent behaviour for 'pass-by-reference' semantics.

A `data.table` can have ALTREP columns.
A `data.tables` will coerce the column to a materialized column when it is modified, even by reference.

Mutability Rules With Respect To Recursive Objects

Lists are difficult objects in that they do not contain elements, they simply point to other objects, that one can access via a list.
When a recursive object is of a mutable class, all its subsets are treated as mutable, as long as they are part of the object.
On the other hand, When a recursive object is of an immutable class, its recursive subsets retain their original mutability.

Example 1: Mutable data.tables
A `data.table` is a mutable class.
So all columns of the `data.table` are treated as mutable;
There is no requirement to, for instance, first change all columns into the class of mutable_atomic

to modify these columns by reference.

Example 2: Immutable lists
A regular `list` is an immutable class.
So the list itself is immutable, but the recursive subsets of the list retain their mutability.
If you have a list of `data.table` objects, for example, the data.tables themselves remain mutable.
Therefore, the following pass-by-reference modification will work without issue:

```
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` has the same address as `x$a`, and is therefore not a copy of `x$a`.
Thus changing `mypointer` also changes `x$a`.
In other words: `mypointer` is what could be called a "**View**" of `x$a`.

Input Variable

Methods/functions that perform in-place modification by reference only works on objects that actually exist as an actual variable, similar to functions in the style of `some_function(x, ...) <- value`.
Thus things like any of the following,
`sb_set(1:10, ...)`, `sb2_set(x$a, ...)`, or `sb_set(base::letters)`,
will not work.

Lock Binding

Mutable classes are, as the name suggests, meant to be mutable.
Locking the binding of a mutable object is **mostly** fruitless (but not completely; see the currentBindings function).
To ensure an object cannot be modified by any of the methods/functions from 'squarebrackets', 2 things must be true:

- the object must be an immutable class.
- the binding must be **locked** (see lockBinding).

Protection

Due to the properties described above in this help page, 'squarebrackets' protects the user from do something like the following:

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

'squarebrackets' will give an error when running the code above, because:

1. most addresses in `baseenv()` are protected;
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.

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
```

---

aaa08_squarebrackets_coercion
Auto-Coercion Rules

Description

This help page describes the auto-coercion rules of the supported classes, as they are handled by the 'squarebrackets' package.

Coercion Rules for Immutable Classes

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`.

Derived From Atomic

coercion_through_copy: depends
Factors, datetime, POSIXct and so on are derived from atomic vectors, but have attributes and special methods that make them behave differently.
Depending on their behaviour, they may or may not allow coercion.
Factors, for example, only accept values that are part of their levels, and thus do not support coercion on modification.

There are highly specialized packages to handle objects derived from atomic objects.
For example the 'anytime' package to handle date-time objects.

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()`).

Coercion Rules for Mutable Classes

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 **(partial) 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.

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_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 View is a data.table, coercion rules of data.tables apply.
And if the View is a mutable_atomic matrix, coercion rules of mutable_atomic 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`)
sb2_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

# sb_set():
sb2_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`)

sb2_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt
  # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
sb2_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`)
sb2_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)
dt_setcoe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb2_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by dt_setcoe(); so no warnings
)
print(obj)


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

# View of List ====

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 <- x$a
address(mypointer) == address(x$a) # they are the same
sb2_set(mypointer, col = "cola", tf = \(x)x^2)
print(x) # notice x has been changed
```

`aaa09_squarebrackets_inconveniences`

Examples Where the Square Bracket Operators Are Less Convenient

---

Description

This help page shows some examples where the square bracket operators (`[`, `[<-`) are less than optimally convenient, and how the methods provided by 'squarebrackets' can be helpful in those cases.

Arrays

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, drop = FALSE]

x[i, j, k] <- value
```

But how would one the use the `[` and `[<-` operators, when number of dimensions of `x` is not known a-priori?
It's not impossible, but still rather convoluted.

The methods provided by 'squarebrackets' do not use position-based arguments, and as such work on any arbitrary dimensions without requiring prior knowledge;
see squarebrackets_indx_args for details.

Rule-sets for data.frame-like Objects

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.

The methods provided by 'squarebrackets' use the same sub-setting rules for all data.frame inherited classes, thus solving this issue.

The 'squarebrackets' 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.

Long Vectors

Performing sub-set operations on a long vector `x` using `[`, like any other object, requires an indexing vector.
The indexing vector may need to be of type `double` (since long vectors can be longer than `2^31 - 1`), and the indexing vector may need to be very large itself also.
This is can be quite inefficient, as one may need up to twice the memory of the object itself.

'squarebrackets' provides the slice_ methods, which can perform sub-set operations on large atomic vectors, **without** the need of any indexing vector at all.

Note that dimensional objects, such as arrays and data.frame-like objects, are in less dire need of a `slice_` method due to dimensional indexing:
For example, a `1500` by `1500` by `1500` array is already a long vector (`1500^3 > (2^31 - 1)`), yet one does not need indexing vectors longer than 1500 elements or of type `double`.

Annoying Sub-setting By Names

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.

The methods provided by 'squarebrackets' uses the same syntax for both selecting and removing sub-sets.
Moreover, selecting/removing sub-sets by names always selects/removes all sub-sets with the given names, not just the first match.

Modification Semantics

'R' adheres to copy-on-modify 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.

The 'squarebrackets' package provides the sb_mod method to return a copy of an object with modified subsets, and the sb_set method to modify using pass-by-reference semantics.
The idx method can be used in combination with R's own `[<-` operator for R's default copy-on-modify semantics.

Regarding Other Packages

There are some packages that solve some of these issues.
But using different packages for solving different issues for the same common theme (in this case: solving some inconveniences in the square bracket operators) leads to inconsistent code.
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.

`bind` *Dimensional Binding of Objects*

Description

The `bind_`implementations provide dimensional binding functionalities.
When possible, the `bind_` functions return mutable classes.

The following implementations are available:

- `bind_mat()` binds dimensionless (atomic/recursive) vectors and (atomic/recursive) matrices row- or column-wise.
  If the result is atomic, returns a mutable_atomic matrix; otherwise returns a recursive matrix.
- `bind_array()` binds (atomic/recursive) arrays and (atomic/recursive) matrices.
  If the result is atomic, returns a mutable_atomic array; otherwise returns a recursive array.
- `bind_dt()` binds data.tables and other data.frame-like objects.
  Returns a `data.table`.
  Faster than `do.call(cbind, ...)` or `do.call(rbind, ...)` for regular `data.frame` objects.

Note that the naming convention of the binding implementations here is "bind_" followed by the **resulting class** (abbreviated).
I.e. `bind_mat` **returns** a matrix, but can bind both matrices and vectors.
And `bind_array` **returns** an array, but can bind both arrays and matrices.
And `bind_dt` **returns** a data.table, but can bind not only data.tables, but also most other data.frame-like objects.

Usage

```
bind_mat(arg.list, along, name_deparse = TRUE, comnames_from = 1L)

bind_array(
  arg.list,
  along,
  name_along = TRUE,
  comnames_from = 1L,
  name_flat = FALSE
)

bind_dt(arg.list, along)
```

Arguments

`arg.list`	a list of only the appropriate objects. If `arg.list` is named, its names will be used for the names of dimension `along` of the output, as far as possible.
`along`	a single integer, indicating the dimension along which to bind the dimensions. I.e. use `along = 1` for row-binding, `along = 2` for column-binding, etc. For arrays, additional flexibility is available: • Specifying `along = 0` will bind the arrays on a new dimension before the first, making `along` the new first dimension. • Specifying `along = n+1`, with `n` being the last available dimension, will create an additional dimension (`n+1`) and bind the arrays along that new dimension.
`name_deparse`	Boolean, for `bind_mat()`. Indicates if dimension `along` should be named. Uses the naming method from rbind/cbind itself.
`comnames_from`	integer scalar or `NULL`, for `bind_array()`. Indicates which object in `arg.list` should be used for naming the shared dimension. If `NULL`, no communal names will be given. For example: When binding columns of matrices, the matrices will share the same rownames. Using `comnames_from = 10` will then result in `bind_array()` using `rownames(arg.list[[10]])` for the rownames of the output.
`name_along`	Boolean, for `bind_array()`. Indicates if dimension `along` should be named.
`name_flat`	Boolean, for `bind_array()`. Indicates if flat indices should be named.

Note that setting this to TRUE will reduce performance considerably.
for performance: set to FALSE

Details

bind_array() is a modified version of the fantastic abind::abind function by Tony Plare & Richard Heiberger (2016), in the following ways:

- bind_array() primarily differs from abind::abind in that it can handle recursive arrays properly
  (the original abind::abind function would unlist everything to atomic arrays, ruining the structure).
- unlike abind::abind, bind_array() only binds (atomic/recursive) arrays and matrices.
  bind_array()does not attempt to convert things to arrays when they are not arrays, but will give an error instead.
  This saves computation time and prevents unexpected results.
- if bind_array() results in an atomic array, it will be a mutable_atomic array.
- bind_array() has more streamlined naming options.

bind_mat() is a modified version of rbind/cbind.
The primary differences is that bind_mat() gives an error when fractional recycling is attempted (like binding 1:3 with 1:10).

Value

The bound 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

```
# bind_array ====

# here, atomic and recursive matrices are mixed,
# resulting in a recursive matrix

# creating the arrays
x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- n(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
```

```
print(y)

# binding the arrays
arg.list <- list(x = x, y = y)
bind_array(arg.list, along = 0L) # binds on new dimension before first
bind_array(arg.list, along = 1L) # binds on first dimension
bind_array(arg.list, along = 2L)
bind_array(arg.list, along = 3L) # bind on new dimension after last



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

# bind_mat ====

# here, atomic and recursive matrices are mixed,
# resulting in a recursive matrix

x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- n(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
print(y)

bind_mat(n(x = x, y = y), 2L)



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

# bind_dt ====


x <- data.frame(a = 1:12, b = month.abb) # data.frame
y <- data.table::data.table(a = 1:12, b = month.abb) # data.table

bind_dt(n(x = x, y = y), 2L) # column bind

bind_dt(n(x = x, y = y), 1L) # row bind
```

`ci_flat` *Construct Indices*

Description

These functions construct flat or dimensional indices.

- `ci_flat()` constructs an integer vector flat indices.
- `ci_margin()` constructs an integer vector of indices for one particular dimension margin.
- `ci_sub()` constructs a list of integer subscripts.
- `ci_df()` is the same as `ci_margin()`, except it is specifically designed for data.frame-like objects.
  It is a separate function, because things like `dimnames(x)[1]` and `rownames(x)` do not always return the same output for certain data.frame-like objects.

Usage

```
ci_flat(
  x,
  i,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)

ci_margin(
  x,
  slice,
  margin,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)

ci_sub(
  x,
  sub,
  dims,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)

ci_df(
  x,
  slice,
  margin,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)
```

Arguments

`x`	the object for which the indices are meant.
`i, slice, margin, sub, dims, inv`	See squarebrackets_indx_args.
`chkdup`	see squarebrackets_options. for performance: set to FALSE
`uniquely_named`	Boolean, indicating if the user knows a-priori that the relevant names of `x` are unique. If set to `TRUE`, speed may increase. But specifying `TRUE` when the relevant names are not unique will result in incorrect output.
`.abortcall`	environment where the error message is passed to.

Value

An integer vector of constructed indices.

Examples

```
x <- matrix(1:25, 5, 5)
colnames(x) <- c("a", "a", "b", "c", "d")
print(x)

bool <- sample(c(TRUE, FALSE), 5, TRUE)
int <- 1:4
chr <- c("a", "a")
cplx <- 1:4 * -1i
tci_bool(bool, nrow(x))
tci_int(int, ncol(x), inv = TRUE)
tci_chr(chr, colnames(x))
tci_cplx(cplx, nrow(x))

ci_flat(x, 1:10 * -1i)
ci_margin(x, 1:4, 2)
ci_sub(x, n(1:5 * -1i, 1:4), 1:2)
```

`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:
The 'squarebrackets' methods and functions that perform modification by reference (basically all methods and functions with "set" in the name) accept `mutable_atomic`, but do 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.

Exposed functions (beside the S3 methods):

- `mutable_atomic()`: create a `mutable_atomic` object from given data.
- `couldb.mutable_atomic()`: checks if an object could become `mutable_atomic`.
  An objects can become `mutable_atomic` if it is one of the following types:
  logical, integer, double, character, complex, raw.
  `bit64::`integer64 type is also supported, since it is internally defined as double.
- `typecast.mutable_atomic()` type-casts and possibly reshapes a (mutable) atomic object, and returns a `mutable_atomic` object.
  Does not preserve dimension names if dimensions are changed.

Usage

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

as.mutable_atomic(x, ...)

## Default S3 method:
as.mutable_atomic(x, ...)

is.mutable_atomic(x)

couldb.mutable_atomic(x)

typecast.mutable_atomic(x, type = typeof(x), dims = dim(x))

## S3 method for class 'mutable_atomic'
c(..., use.names = TRUE)

## 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

`data`	atomic vector giving data to fill the `mutable_atomic` object.
`names, dim, dimnames`	see setNames and array.

x	an atomic object.
...	method dependent arguments.
type	a string giving the type; see typeof.
dims	integer vector, giving the new dimensions.
use.names	Boolean, indicating if names should be preserved.
value	see Extract.

Value

For `mutable_atomic()`, `as.mutable_atomic()`, `typecast.mutable_atomic()`:
Returns a `mutable_atomic` object.

For `is.mutable_atomic()`:
Returns `TRUE` if the object is `mutable_atomic`, and returns `FALSE` otherwise.

For `couldb.mutable_atomic()`:
Returns `TRUE` if the object is one of the following types:
logical, integer, double, character, complex, raw.
`bit64::`integer64 type is also supported, since it is internally defined as double.
Returns `FALSE` otherwise.

Warning

Always use the exported functions given by 'squarebrackets' to create a `mutable_atomic` object, as they make necessary checks.
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
typecast.mutable_atomic(x, "character")

x <- matrix(1:10, ncol = 2)
x <- as.mutable_atomic(x)
is.mutable_atomic(x)
print(x)
x[, 1]
x[] <- as.double(x)
print(x)
is.mutable_atomic(x)
```

`cp_seq` *Construct Parameters for a Sequence Based on Margins*

Description

`cp_seq()` returns a list of parameters to construct a sequence based on the margins of an object. It is internally used by the idx_r function and slice method.

Usage

```
cp_seq(x, m = 0L, from = NULL, to = NULL, by = 1L)
```

Arguments

`x`	the object for which to compute margin-based sequence parameters.
`m`	integer or complex, giving the margin(s). For non-dimensional objects or for flat indices, specify `m = 0L`.
`from`	integer or complex, of the same length as `m` or of length 1, specifying the from point.
`to`	integer or complex, of the same length as `m` or of length 1, specifying the **maximally allowed** end value.
`by`	integer, of the same length as `m` or of length 1, specifying the step size.

Value

A list of the following elements:

`$start`:
The actual starting point of the sequence.
This is simply `from` translated to regular numeric.

`$end`:
The **actual** ending point of the sequence.
This is **not** the same as `to`, not even when translated to regular numeric.
For example, the following code:

```
seq(from = 1L, to = 10L, by = 2L)
#> [1] 1 3 5 7 9
```

specifies `to = 10L`.
But the sequence doesn't actually end at `10`; it ends at `9`.
Therefore, `cp_seq(x, m, 1, 10, 2)` will return `end = 9`, not `end = 10`.
This allows the user to easily predict where an sequence given in idx_r/slice will actually end.

`$by`:
This will give `by`, but with it's sign adjusted, if needed.

`$length.out`:
The actual vector lengths the sequences would be, given the translated parameters.

Arguments Details

Multiple dimensions at once
The `cp_seq` function can construct the sequence parameters needed for multiple dimensions at once, by specifying a vector for `m`.
The lengths of the other arguments are then recycled if needed.

Using only `by`
If `from, to` are not specified, using `by` will construct the following sequence:
If `by` is positive, `seq.int(1L, n, by)`.
If `by` is negative, `seq.int(n, 1L, by)`.
Where `n` is the maximum index (i.e. `length(x)` or `dim(x)[m]`, depending on the situation).

Using `from, to, by`
If `from, to, by` are all specified, `by` is stored as `abs(by)`, and the sign of `by` is automatically adjusted to ensure a sensible sequence is created.

Examples

```
x <- data.frame(
  a = 1:10, b = letters[1:10], c = factor(letters[1:10]), d = -1:-10
)
print(x)
ind1 <- idx_r(x, 1, 2, 2* -1i) # rows 2:(nrow(x)-1)
sb2_x(x, ind1) # extract the row range

x <- array(1:125, c(5,5,5))
dims <- 1:3
sub <- idx_r(x, dims, 2, 2* -1i) # 2:(n-1) for every dimension
sb_x(x, sub, dims) # same as x[ 2:4, 2:4, 2:4, drop = FALSE]

x <- letters
x[idx_r(x, 0, 2, 2* -1i)]
```

`currentBindings` *List or Lock All Currently Existing Bindings Pointing To Same Address*

Description

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

`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.

`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

```
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: • `"list"` (default). • `"checklock"`. • `"lockbindings"`.
`env`	the environment where to look for objects. If `NULL` (default), the caller environment is used.

Details

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:

```
x <- mutable_atomic(1:16)
y <- x
lockBinding("y", environment())
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()`/`sb2_set()` performs modification by reference, `y` will still be modified, despite being locked.

The `currentBindings()` function allows to user to: find all **currently existing** bindings in the **caller environment** sharing the same address as `x`, and locking all these bindings.

Value

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

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

Warning

The `currentBindings()` function only locks **currently existing** bindings in the **specified environment**;
bindings that are created **after** calling `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, `currentBindings()` must be called again.

Examples

```
x <- as.mutable_atomic(1:10)
y <- x
lockBinding("y", environment())
currentBindings(x)
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
currentBindings(x, "lockbindings") # lock all
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!
currentBindings(x, "checklock") # z is not locked
currentBindings(x, "lockbindings") # we must re-run this
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
```

`dt` *Functional Forms of data.table Operations*

Description

Functional forms of special data.table operations.
These functions do not use Non-Standard Evaluation.
These functions also benefit from the security measures that 'squarebrackets' implements for the pass-by-reference semantics.

- `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/tidytable `new` to data.table/tidytable `x`, thereby modifying `x` using pass-by-reference semantics.
- `dt_setreorder()` reorders the rows and/or variables of a `data.table` using pass-by-reference semantics.

Usage

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

```
dt_setcoe(
  x,
  col = NULL,
  vars = NULL,
  v,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

dt_setrm(
  x,
  col = NULL,
  vars = NULL,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

dt_setadd(x, new)

dt_setreorder(x, roworder = NULL, varorder = NULL)
```

Arguments

`x`	a `data.table` or `tidytable`.
`SDcols`	atomic vector, giving the columns to which the aggregation function `f()` is to be applied on.
`f`	the aggregation function
`by`	atomic vector, giving the grouping columns.
`order_by`	Boolean, indicating if the aggregated result should be ordered by the columns specified in `by`.
`col`, `vars`	see squarebrackets_indx_args. Duplicates are not allowed.
`v`	the coercive transformation function
`chkdup`	see squarebrackets_options. for performance: set to FALSE
`new`	a data.frame-like object. It must have column names that do not already exist in `x`.
`roworder`	a integer vector of the same length as `nrow(x)`, giving the order in which the rows are to be re-order. Internally, this numeric vector will be turned into an order using order, thus ensuring it is a strict permutation of `1:nrow(x)`.
`varorder`	integer or character vector of the same length as `ncol(x)`, giving the new column order. See `data.table::`setcolorder.

Details

`dt_setreorder(x, roworder = roworder)` internally creates a new column to reorder the data.table by, and then removes the new column.
The column name is randomized, and extra care is given to ensure it does not overwrite any existing columns.

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 assignments like `x <- dt_setcoe(x, ...)`.
Since these functions return void, you'll just get `NULL`.

Examples

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

if(requireNamespace("sf")) {
  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`)
sb2_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, v = as.numeric) # integers are now numeric
str(obj)
sb2_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)



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


# dt_setreorder====

n <- 1e4
obj <- data.table::data.table(
  a = 1L:n, b = n:1L, c = as.double(1:n), d = as.double(n:1)
)
dt_setreorder(obj, roworder = n:1)
head(obj)
dt_setreorder(obj, varorder = ncol(obj):1)
head(obj)
```

idx *Convert/Translate Indices (for Copy-On-Modify Substitution)*

Description

The `idx()` method converts indices.
The type of output depends on the type of input index arguments given:

- `idx(x, i = i, ...)` converts linear indices to a strictly positive integer vector of linear indices.
- `idx(x, sub = sub, dims = dims, ...)` converts dimensional indices to a strictly positive integer vector of linear indices.
- `idx(x, slice = slice, margin = margin, ...)` converts indices of one dimension to a strictly positive integer vector of indices for that specific dimension.

Vectors (both atomic and recursive) only have index argument `i`.
Data.frame-like objects only have the `slice, margin` index argument pair.
Arrays (both atomic and recursive) have the `sub, dims` index argument pair, as well as the arguments `i` and `slice, margin`.

The result of the `idx()` method can be used inside the regular square-brackets operators.
For example like so:

```
x <- array(...)
my_sub2ind <- idx(x, sub, dims)
x[my_sub2ind] <- value

y <- data.frame(...)
rows <- idx(y, 1:10, 1, inv = TRUE)
cols <- idx(y, c("a", "b"), 2)
y[rows, cols] <- value
```

thus allowing the user to benefit from the convenient index translations from 'squarebrackets', whilst still using R's default copy-on-modification semantics (instead of the semantics provided by 'squarebrackets').

Usage

```
idx(x, ...)

## Default S3 method:
idx(x, i, inv = FALSE, ..., chkdup = getOption("squarebrackets.chkdup", FALSE))

## S3 method for class 'array'
idx(
  x,
  sub = NULL,
  dims = 1:ndims(x),
  slice = NULL,
  margin = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'data.frame'
idx(
```

```
  x,
  slice,
  margin,
  inv = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
```

Arguments

`x`	vector, matrix, array, or data.frame; both atomic and recursive objects are supported.
`...`	see squarebrackets_method_dispatch.
`i, sub, dims, margin, slice, inv`	See squarebrackets_indx_args. Duplicates are not allowed.
`chkdup`	see squarebrackets_options. for performance: set to FALSE

Value

For `idx(x, i = i, ...)` and `idx(x, sub = sub, dims = dims, ...)`:
A strictly positive integer vector of flat indices.

For `idx(x, margin = margin, slice = slice, ...)`:
A strictly positive integer vector of indices for the dimension specified in `margin`.

Examples

```
# atomic ====

x <- 1:10
x[idx(x, \(x)x>5)] <- -5
print(x)

x <- array(1:27, dim = c(3,3,3))
x[idx(x, n(1:2, 1:2), c(1,3))] <- -10
print(x)


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


# recursive ====

x <- as.list(1:10)
x[idx(x, \(x)x>5)] <- -5
print(x)

x <- array(as.list(1:27), dim = c(3,3,3))
x[idx(x, n(1:2, 1:2), c(1,3))] <- -10
```

```
print(x)


x <- data.frame(
  a = sample(c(TRUE, FALSE, NA), 10, TRUE),
  b = 1:10,
  c = rnorm(10),
  d = letters[1:10],
  e = factor(letters[11:20])
)
rows <- idx(x, 1:5, 1, inv = TRUE)
cols <- idx(x, c("b", "a"), 2)
x[rows, cols] <- NA
print(x)
```

---

idx_by *Compute Grouped Indices*

Description

Given:

- a sub-set function `f`;
- an object `x` with its margin `m`;
- and a grouping factor `grp`;

the `idx_by()` function takes `indices` **per group** `grp`.
The result of `idx_by()` can be supplied to the indexing arguments (see squarebrackets_indx_args) to perform **grouped** subset operations.

Usage

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

Arguments

`x`	the object from which to compute the indices.
`m`	a single non-negative integer giving the margin for which to compute indices. For flat indices or for non-dimensional objects, use `m = 0L`.
`f`	a subset function to be applied per group on `indices`. If `m == 0L`, `indices` is here defined as `setNames(1:length(x), names(x))`. If `m > 0L`, `indices` is here defined as `setNames(1:dim(x)[m], dimnames(x)[[m]])`. The function must produce a character or integer vector as output. For example, to subset the last element per group, specify: `f = last`
`grp`	a factor giving the groups.
`parallel, mc.cores`	see BY.

Value

A vector of indices.

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(a, 0L, last, grp)
sb_x(cbind(a, grp), i, 1L)


# 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(x, 1, first, x$group)
sb2_x(x, row)
# get the first row for each group for which a > 10:
x2 <- sb2_x(x, filter = ~ a > 10)
row <- na.omit(idx_by(x2, 1, first, x2$group))
sb2_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 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 = 1L` will cut `x` into individual rows, and apply the order on those rows. And `margin = 2L` 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))
```

idx_r	*Compute Integer Index Range*

Description

`idx_r()` computes integer index range(s).

Usage

```
idx_r(x, m = 0L, from = NULL, to = NULL, by = 1L)
```

Arguments

`x`	the object for which to compute subset indices.
`m`, `from`, `to`, `by`	see cp_seq.

Value

If `length(m) == 1L`: a vector of numeric indices.

If `length(m) > 1L`: a list of the same length as `m`, containing numeric vectors of indices.

Examples

```
x <- data.frame(
  a = 1:10, b = letters[1:10], c = factor(letters[1:10]), d = -1:-10
)
print(x)
ind1 <- idx_r(x, 1, 2, 2* -1i) # rows 2:(nrow(x)-1)
sb2_x(x, ind1) # extract the row range

x <- array(1:125, c(5,5,5))
dims <- 1:3
sub <- idx_r(x, dims, 2, 2* -1i) # 2:(n-1) for every dimension
sb_x(x, sub, dims) # same as x[ 2:4, 2:4, 2:4, drop = FALSE]

x <- letters
x[idx_r(x, 0, 2, 2* -1i)]
```

indx_x *Exported Utilities*

Description

Exported utilities.
Usually the user won't need these functions.

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))
```

lst *Unnest Tree-like List into a Recursive Matrix or Flattened Recursive Vector*

Description

`[[`, `[[<-`, sb2_rec, and sb2_recin, can perform recursive subset operations on a nested list.
Such recursive subset operations only operate on a single element.
Performing recursive subset operations on multiple elements is not vectorized, and requires a (potentially slow) loop.

The `lst_untree()` function takes a nested tree-like list, and turns it into a recursive matrix (a matrix of list-elements), allowing vectorized subset operations to be performed on the nested list.

`lst_untree()` can also simply flatten the list, making it a non-nested list.
See the Examples section to understand how the list will be arranged and named.

The `lst_nlists()` counts the total number of recursive list-elements inside a list.

Usage

```
lst_nlists(x)

lst_untree(x, margin, use.names = TRUE)
```

Arguments

`x` a tree-like nested list.

`margin` a single integer, indicating how the result should be arranged:

- `margin = 0` produces a simple flattened recursive vector (i.e. list) without dimensions.
- `margin = 1` produces a recursive matrix (i.e. a matrix of list-elements), with `length(x)` rows and `n` columns,
  where `n = sapply(x, lst_nlists) |> max()`.
  Empty elements will be filled with `list(NULL)`.
- `margin = 2` produces a recursive matrix (i.e. a matrix of list-elements), with `length(x)` columns and `n` rows,
  where `n = sapply(x, lst_nlists) |> max()`.
  Empty elements will be filled with `list(NULL)`.

`use.names` Boolean, indicating if the result should be named.
See section "use.names" for more information.

Value

For `lst_untree()`:
A non-nested (dimensional) list.
Note that if `margin = 1` or `margin = 2`, `lst_untree()` returns a recursive matrix (i.e. a recursive array with 2 dimensions), **not** a data.frame.
To turn a nested list into a data.frame instead, one option would be to use:
`rrapply(x, how = "melt")`

For `lst_nlists()`:
A single integer, giving the total number of recursive list-elements in the given list.

use.names

`margin = 0` **and** `use.names = TRUE`
If `margin = 0` and `use.names = TRUE`, every element in the flattened list will be named.
Names of nested elements, such as `x[["A"]][["B"]][["C"]]`, will become `"A.B.C"`, as that is the behaviour of the rapply function (which `lst_untree()` calls internally).
It is therefore advised not to use dots (`"."`) in your list names, and use underscores (`"_"`) instead, before calling `lst_untree()`.

See the `rrapply::rrapply` function for renaming (and other forms of transforming) recursive subsets of lists.

`margin = 1` **and** `use.names = TRUE`
If `margin == 1` and `use.names = TRUE`, the rows of resulting recursive matrix will be equal to `names(x)`, but recursive names will not be assigned.

`margin = 2` **and** `use.names = TRUE`
If `margin == 2` and `use.names = TRUE`, the columns of resulting recursive matrix will be equal to `names(x)`, but recursive names will not be assigned.

`use.names = FALSE`
If `use.names = FALSE`, the result will not have any names assigned at all.

Examples

```
# show-casing how the list-elements are arranged and named ====

x <- list(
  A = list(
    A = list(A = "AAA", B = "AAB"),
    A = list(A  = "AA2A", B = "AA2B"),
    B = list(A = "ABA", B = "ABB"),
    C = letters
  ),
  Y = list(
    Z = list(Z = "YZZ", Y = "YZY"),
    Y = list(Z = "YYZ", Y = "YYY"),
    X = "YX"
  )
)


# un-tree column-wise:
sapply(x, lst_nlists) |> max() # number of rows `y` will have
y <- lst_untree(x, margin = 2L, use.names = TRUE)
dim(y)
print(y)
sb2_x(y, n(1:3, 1:2), 1:ndims(y)) # vectorized selection of multiple recursive elements


# un-tree row-wise:
sapply(x, lst_nlists) |> max() # number of columns `y` will have
y <- lst_untree(x, margin = 1L, use.names = TRUE)
dim(y)
print(y)
sb2_x(y, n(1:2, 1:3), 1:ndims(y))  # vectorized selection of multiple recursive elements


# simple flattened list:
y <- lst_untree(x, margin = 0, use.names = TRUE)
print(y)
y[["Y.Z.Y"]]
x[[c("Y", "Z", "Y")]] # equivalent in the original list
```

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

# showcasing that only list-elements are recursively flattened ====
# i.e. atomic vectors in recursive subsets remain atomic

x <- lapply(1:10, \(x)list(sample(letters), sample(1:10)))

sapply(x, lst_nlists) |> max()
y <- lst_untree(x, margin = 1)
dim(y)
print(y)

lst_untree(x, margin = 1)


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

# showcasing vectorized sub-setting ====
x <- lapply(1:10, \(x) list(
  list(sample(letters[1:10]), sample(LETTERS[1:10])),
  list(sample(month.abb), sample(month.name)),
  list(sample(1:10), rnorm(10))
))
y <- lst_untree(x, 1)

# getting the first recursive elements in the second level/depth in base R:
for(i in seq_along(x)) {
  x[[c(i, c(1L, 1L))]] |> print() # for-loop, slow
}

# the same, but vectorized using the untree'd list:
sb2_x(y, n(1:nrow(y), 1L), 1:ndims(y)) |> drop() |> print() # vectorized, fast
```

`match_all` *Match All, Order-Sensitive and Duplicates-Sensitive*

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.

`match_all()` is essentially a much more efficient version of:

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

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

`match_all()` internally calls `collapse::`fmatch and `collapse::`gsplit.
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 of the same type.
`needles` cannot contain `NA`/`NaN`.
Long vectors are not supported.

`unlist`
: Boolean, indicating if the result should be a single unnamed integer vector (`TRUE`, default), or a named list of integer vectors (`FALSE`).

Value

An integer vector, or list of integer vectors.
If a list, each element of the list corresponds to each value of `needles`.
When `needles` and/or `haystack` is empty, or when `haystack` is 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)
```

---

ma_setv — *Find and Replace Present Values in mutable_atomic Objects By Reference*

Description

The `ma_setv(x, v rp)` function performs the equivalent of
`x[which(x == v)] <- rp`
but using pass-by-reference semantics.

This is faster than using `sb_set(x, i = which(x == v), rp = rp)`.

Inspired by `collapse::`setv, but written in 'C++' through 'Rcpp', with additional safety checks.

Usage

```
ma_setv(x, v, rp, invert = FALSE, NA.safety = TRUE)
```

Arguments

x	a mutable_atomic **variable**.
v	non-missing (so no NA or NaN) atomic scalar to find.
rp	atomic scalar giving the replacement value.
invert	Boolean. If FALSE (default), the equivalent of `x[which(x == v()] <- rp` is performed; If TRUE, the equivalent of `x[which(x != v)] <- rp` is performed instead.
NA.safety	Boolean. just like in which, NA and NaN results in x==v should be ignored, thus NA.safety is TRUE by default. However, if it is known that x contains no NAs or NaNs, setting NA.safety to FALSE will increase performance a bit. NOTE: Setting NA.safety = FALSE when x does contain NAs or NaNs, may result in unexpected behaviour. for performance: set to FALSE

Value

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

Examples

```
x <- mutable_atomic(c(1:20, NA, NaN))
print(x)
ma_setv(x, 2, 100)
print(x)
```

---

n *Nest*

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 **nests** 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]
```

---

ndims *Get Number of Dimensions*

Description

`ndims(x)` is short-hand for `length(dim(x))`.

Usage

```
ndims(x)
```

Arguments

`x`	the object to get the number of dimensions from.

Value

An integer, giving the number of dimensions `x` has.
For vectors, gives `0L`.

Examples

```
x <- 1:10
ndims(x)
obj <- array(1:64, c(4,4,3))
print(obj)
ndims(obj)
```

sb2_rec — *Access, Replace, Transform, Remove, or Extend Recursive Subsets*

Description

The sb2_rec() and sb2_recin() methods are essentially convenient wrappers around [[and [[<-, respectively.
Unlike [[and [[<-, these are actually S3 methods, so package authors can create additional method dispatches.

sb2_rec() will access recursive subsets of lists.

sb2_recin() can do the following things:

- replace or transform recursive subsets of a list, using R's default Copy-On-Modify semantics, by specifying the rp or tf argument, respectively.
- remove a recursive subset of a list, using R's default Copy-On-Modify semantics, by specifying argument rp = NULL.
- extending a list with additional recursive elements, using R's default Copy-On-Modify semantics.
  This is done by specifying an out-of-bounds index in argument rec, and entering the new values in argument rp.
  Note that adding surface level elements of a dimensional list will remove the dimension attributes of that list.

Usage

```
sb2_rec(x, ...)

## Default S3 method:
sb2_rec(x, rec, ...)

sb2_recin(x, ...)

## Default S3 method:
sb2_recin(x, rec, ..., rp, tf)
```

Arguments

x	a list, or list-like object.
...	see squarebrackets_method_dispatch.
rec	a strictly positive integer vector or character vector, of length p, such that sb2_rec(x, rec) is equivalent to x[[rec[1]]]...[[rec[p]]], 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, since recursive indexing can only select a single element.

rp — optional, and allows for multiple functionalities:

- In the simplest case, performs `x[[rec]] <- rp`, using R's default semantics. Since this is a replacement of a recursive subset, `rp` does not necessarily have to be a list itself;
  `rp` can be any type of object.
- Specifying `rp = NULL` will **remove** (recursive) subset `sb(x, rec)`.
  To specify actual `NULL` instead of removing a subset, use `rp = list(NULL)`.
- When `rec` is an integer, and specifies an out-of-bounds subset, `sb2_recin()` will add value `rp` to the list.
  Any empty positions in between will be filled with `NA`.
- When `rec` is character, and specifies a non-existing name, `sb2_recin()` will add value `rp` to the list as a new element at the end.

tf — an optional function. If specified, performs `x[[rec]] <- tf(x[[rec]])`, using R's default Copy-On-Modify semantics.
Does not support extending a list like argument `rp`.

Details

Since recursive objects are references to other objects, extending a list or removing an element of a list does not copy the entire list, in contrast to atomic vectors.

Value

For `sb2_rec()`:
Returns the recursive subset.

For `sb2_recin(..., rp = rp)`:
Returns VOID, but replaces, adds, or removes the specified recursive subset, using R's default Copy-On-Modify semantics.

For `sb2_recin(..., tf = tf)`:
Returns VOID, but transforms the specified recursive subset, using R's default Copy-On-Modify 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 ====

sb2_rec(lst, c(1,2,2)) # this gives "AA2B"
sb2_rec(lst, c("A", "B", "B")) # this gives "ABB"
sb2_rec(lst, c(2,2,1)) # this gives "BBA"
sb2_rec(lst, c("B", "B", "A")) # this gives "BBA"


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

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

# replace "AAB" using R's default in-place semantics:
sb2_recin(
  lst, c("A", "A", "B"),
  rp = "THIS IS REPLACED WITH IN-PLACE SEMANTICS"
)
print(lst)



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

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

for(i in c("A", "B")) sb2_recin(lst, i, rp = "AND THEN THERE WERE NONE")

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 <- sb2_rec(x, "a")
address(mypointer) == address(x$a) # they are the same
sb2_set(mypointer, col = "cola", tf = \(x)x^2)
print(x) # notice x has been changed
```

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.
Use `sb_mod(x, ...)` if `x` is an atomic object; this returns a full copy.

Use `sb2_mod(x, ...)` if `x` is a recursive object (i.e. list or data.frame-like); this returns a partial copy.

For modifying subsets using R's default copy-on-modification semantics, see idx.

Usage

```
sb_mod(x, ...)

## Default S3 method:
sb_mod(
  x,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'array'
sb_mod(
  x,
  sub = NULL,
  dims = 1:ndims(x),
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

sb2_mod(x, ...)

## Default S3 method:
sb2_mod(
  x,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)

## S3 method for class 'array'
sb2_mod(
  x,
  sub = NULL,
```

```
  dims = 1:ndims(x),
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)

## S3 method for class 'data.frame'
sb2_mod(
  x,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  inv = FALSE,
  coe = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)
```

Arguments

- `x` — see squarebrackets_supported_structures.
- `...` — see squarebrackets_method_dispatch.
- `i`, `row`, `col`, `sub`, `dims`, `filter`, `vars`, `inv` — See squarebrackets_indx_args.
  An empty index selection returns the original object unchanged.
- `rp`, `tf`, `.lapply` — see squarebrackets_modify.
- `chkdup` — see squarebrackets_options.
  for performance: set to FALSE
- `coe` — Either `FALSE` (default), `TRUE`, or a function.
  The argument `coe` is ignored if both the `row` and `filter` arguments are set to `NULL`.
  See Details section for more info.
  for performance: set to FALSE

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.

Argument coe
For data.frame-like objects, sb_mod() can only auto-coerce whole columns, not subsets of columns.
So it does not automatically coerce column types when row or filter is also specified.
The coe arguments provides 2 ways to circumvent this:

1. The user can supply a coercion function to argument coe.
   The function is 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()).
2. The user can set coe = TRUE.
   In this case, the whole columns specified in col or vars are extracted and copied to a list.
   Subsets of each list element, corresponding to the selected rows, are modified with rp or tf(), using R's regular auto-coercion rules.
   The modified list is then returned to the data.frame-like object, replacing the original columns.

Note that coercion required additional memory.
The larger the data.frame-like object, the larger the memory.
The default, coe = FALSE, uses the least amount of memory.

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:ndims(obj), 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, "a", 2L, rp = -1:-8)
# above is equivalent to  obj[, which(colnames(obj) %in% "a")] <- -1:-8; obj
sb_mod(obj, 1:3, 1:ndims(obj),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 <- array(1:64, c(4,4,3))
print(obj)
sb_mod(obj, n(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)
sb2_mod(obj, "a", rp = list(1L))
# above is equivalent to  obj[["a"]] <- 1L; obj
sb2_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

obj <- rbind(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_mod(obj, 1:3, 1:ndims(obj),rp = n(-1))
# above is equivalent to obj[1:3, 1:3] <- list(-1)
sb2_mod(obj, i = is.numeric, rp = n(-1))
# above is equivalent to obj[sapply(obj, is.numeric)] <- list(-1)
sb2_mod(obj, "a", 2L, rp = n(-1))
# above is equivalent to
# obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()] <- list(-1)


obj <- array(as.list(1:64), c(4,4,3))
print(obj)
sb2_mod(obj, n(1:3, 1:2), c(1,3), rp = as.list(-1:-24))
# above is equivalent to obj[1:3, , 1:2] <- as.list(-1:-24)
sb2_mod(obj, i = \(x) x <= 5, rp = as.list(-1:-5))
# above is equivalent to obj[sapply(onj, \(x) x <= 5)] <- as.list(-1:-5)


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

# data.frame-like objects  - 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`)
sb2_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

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

# data.frame-like objects  - partial 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`)

sb2_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
sb2_mod(
```

```
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  coe = as.double, tf = sqrt # SAFE: coercion performed
)
sb2_mod(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  coe = TRUE, tf = sqrt # SAFE: coercion performed
)
```

---

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.
Use `sb_rm(x, ...)` if `x` is an atomic object.
Use `sb2_rm(x, ...)` if `x` is a recursive object (i.e. list or data.frame-like).

Usage

```
sb_rm(x, ...)

## Default S3 method:
sb_rm(x, i = NULL, ..., chkdup = getOption("squarebrackets.chkdup", FALSE))

## S3 method for class 'array'
sb_rm(
  x,
  sub = NULL,
  dims = 1:ndims(x),
  i = NULL,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

sb2_rm(x, ...)

## Default S3 method:
sb2_rm(
  x,
  i = NULL,
  red = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'array'
```

```
sb2_rm(
  x,
  sub = NULL,
  dims = 1:ndims(x),
  i = NULL,
  red = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'data.frame'
sb2_rm(
  x,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
```

Arguments

`x` see squarebrackets_supported_structures.

`...` see squarebrackets_method_dispatch.

`i, row, col, sub, dims, filter, vars`
See squarebrackets_indx_args.
An empty index selection results in nothing being removed, and the entire object is returned.

`chkdup` see squarebrackets_options.
for performance: set to FALSE

`red` Boolean, for recursive objects only, indicating if the result should be reduced.
If `red = TRUE`, selecting a single element will give the simplified result, like using `[[]]`.
If `red = FALSE`, a list is always returned regardless of the number of elements.

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:ndims(obj))
# 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, "a", 2L)
# 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))
# above is 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)
sb2_rm(obj, "a")
# above is equivalent to obj[which(!names(obj) %in% "a")]
sb2_rm(obj, 1) # obj[-1]
sb2_rm(obj, 1:2)
# above is equivalent to obj[seq_len(length(obj))[-1:-2]]
sb2_rm(obj, is.numeric, red = 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)
sb2_rm(obj, is.numeric)
# above is equivalent to obj[!sapply(obj, is.numeric)] # this time singular brackets?
# for recusive indexing, see sb2_rec()


obj <- rbind(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_rm(obj, 1:3, 1:ndims(obj))
# above is equivalent to obj[1:3, 1:3, drop = FALSE]
sb2_rm(obj, i = is.numeric)
# above is equivalent to obj[sapply(obj, is.numeric)]
sb2_rm(obj, c("a", "a"), 2L)
# above is equivalent to obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()]

obj <- array(as.list(1:64), c(4,4,3))
print(obj)
sb2_rm(obj, n(1, c(1, 3)), c(1, 3))
# above is equivalent to obj[-1, , c(-1, -3), drop = FALSE]
sb2_rm(obj, i = \(x)x>5)
# above is equivalent to obj[!sapply(obj, \(x) x > 5)]
```

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

# data.frame-like objects ====

obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb2_rm(obj, 1:3, 1:3)
# above is equivalent to obj[-1:-3, -1:-3, drop = FALSE]
sb2_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
Use `sb_set(x, ...)` if x is an atomic object (i.e. mutable_atomic).
Use `sb2_set(x, ...)` if x is a recursive object (i.e. data.table).

Usage

```
sb_set(x, ...)

## Default S3 method:
sb_set(
  x,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'array'
sb_set(
  x,
  sub = NULL,
  dims = 1:ndims(x),
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
```

```
sb2_set(x, ...)

## Default S3 method:
sb2_set(x, ...)

## S3 method for class 'data.table'
sb2_set(
  x,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)
```

Arguments

`x`	a **variable** belonging to one of the supported mutable classes.
`...`	see squarebrackets_method_dispatch.
`i`, `row`, `col`, `sub`, `dims`, `filter`, `vars`, `inv`	See squarebrackets_indx_args. An empty index selection leaves the original object unchanged.
`rp`, `tf`, `.lapply`	see squarebrackets_modify.
`chkdup`	see squarebrackets_options. for performance: set to FALSE

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.

Value

Returns: VOID. This method modifies the object by reference.
Do not use assignments 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()
print(obj)

sb_set(obj, 1:3, 1:ndims(obj), rp = -1:-9)
print(obj2)
# above is like x[1:3, 1:3] <- -1:-9, but using pass-by-reference

obj <- obj2 <- gen_mat()
obj

sb_set(obj, i = \(x) x <= 5, rp = -1:-5)
print(obj2)
# above is like x[x <= 5] <- -1:-5, but using pass-by-reference

obj <- obj2 <- gen_mat()
obj

sb_set(obj, "a", 2L, rp = cbind(-1:-4, -5:-8))
print(obj2)
# above is like x[, "a"] <- cbind(-1:-4, -5:-8), but using pass-by-reference

obj <- obj2 <- gen_mat()
obj

sb_set(obj, 1:3, 1:ndims(obj), tf = \(x) -x)
print(obj2)
# above is like x[1:3, 1:3] <- -1 * x[1:3, 1:3], but using pass-by-reference

obj <- obj2 <- gen_mat()
obj

sb_set(obj, i = \(x) x <= 5, tf = \(x) -x)
print(obj2)
# above is like x[x <= 5] <- -1 * x[x <= 5], but using pass-by-reference

obj <- obj2 <- gen_mat()
obj

sb_set(obj, "a", 2L, tf = \(x) -x)
obj2
# above is like x[, "a"] <- -1 * x[, "a"], but using pass-by-reference


gen_array <- function() {
  as.mutable_atomic(array(1:64, c(4,4,3)))
}
```

```
obj <- obj2 <- gen_array()
obj

sb_set(obj, n(1:3, 1:2, c(1, 3)), 1:3, rp = -1:-12)
print(obj2)
# above is like x[1:3, , 1:2] <- -1:-12, but using pass-by-reference


obj <- obj2 <- gen_array()
obj
sb_set(obj, i = \(x)x <= 5, rp = -1:-5)
print(obj2)
# above is like x[x <= 5] <- -1:-5, but using pass-by-reference



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

# 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`)
sb2_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]))
dt_setcoe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb2_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by dt_setcoe(); 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`)
sb2_set(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)
str(obj)
```

`sb_setRename` *Safely Change the Names of a Mutable Object By Reference*

Description

Functions to rename a supported mutable object using pass-by-reference semantics:

- `sb_setFlatnames()` renames the (flat) names of a `mutable_atomic` object.
- `sb_setDimnames()` renames the dimension names of a `mutable_atomic` object.

- sb2_setVarnames() renames the variable names of a data.table object.

Usage

```
sb_setFlatnames(x, i = NULL, newnames, ...)

sb_setDimnames(x, m, newdimnames, ...)

sb2_setVarnames(x, old, new, skip_absent = FALSE, ...)
```

Arguments

x	a **variable** belonging to one of the supported mutable classes.
i	logical, numeric, character, or imaginary indices, indicating which flatnames should be changed. If i = NULL, the names will be completely replaced.
newnames	Atomic character vector giving the new names. Specifying NULL will remove the names.
...	see squarebrackets_method_dispatch.
m	integer vector giving the margin(s) for which to change the names (m = 1L for rows, m = 2L for columns, etc.).
newdimnames	a list of the same length as m. The first element of the list corresponds to margin m[1], the second element to m[2], and so on. The components of the list can be either NULL, or a character vector with the same length as the corresponding dimension. Instead of a list, simply NULL can be specified, which will remove the dimnames completely.
old	the old column names
new	the new column names, in the same order as old
skip_absent	Skip items in old that are missing (i.e. absent) in names(x). Default FALSE halts with error if any are missing.

Value

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

Examples

```
# mutable atomic vector ====
x <- y <- mutable_atomic(1:10, names = letters[1:10])
```

```
print(x)
sb_setFlatnames(x, newnames = rev(letters[1:10]))
print(y)

x <- y <- mutable_atomic(1:10, names = letters[1:10])
print(x)
sb_setFlatnames(x, 1L, "XXX")
print(y)

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


# mutable atomic matrix ====
x <- mutable_atomic(
  1:20, dim = c(5, 4), dimnames = n(letters[1:5], letters[1:4])
)
print(x)
sb_setDimnames(
  x,
  1:2,
  lapply(dimnames(x), rev)
)
print(x)



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



# data.table ====

x <- data.table::data.table(
  a = 1:20,
  b = letters[1:20]
)
print(x)
sb2_setVarnames(x, old = names(x), new = rev(names(x)))
print(x)
```

---

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. repeat x times) subsets of an object.
Use `sb_x(x, ...)` if x is an atomic object.
Use `sb2_x(x, ...)` if x is a recursive object (i.e. list or data.frame-like).

Usage

```
sb_x(x, ...)

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

## S3 method for class 'array'
sb_x(x, sub = NULL, dims = 1:ndims(x), i = NULL, ...)

sb2_x(x, ...)

## Default S3 method:
sb2_x(x, i = NULL, red = FALSE, ...)

## S3 method for class 'array'
sb2_x(x, sub = NULL, dims = 1:ndims(x), i = NULL, red = FALSE, ...)

## S3 method for class 'data.frame'
sb2_x(x, row = NULL, col = NULL, filter = NULL, vars = NULL, ...)
```

Arguments

`x` see squarebrackets_supported_structures.

`...` see squarebrackets_method_dispatch.

`i, row, col, sub, dims, filter, vars`
See squarebrackets_indx_args.
Duplicates are allowed, resulting in duplicated indices.
An empty index selection results in an empty object of length 0.

`red` Boolean, for recursive objects only, indicating if the result should be reduced.
If `red = TRUE`, selecting a single element will give the simplified result, like using `[[]]`.
If `red = FALSE`, a list is always returned regardless of the number of elements.

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:ndims(obj))
# 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, c("a", "a"), 2L)
# 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))
# above is 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)
sb2_x(obj, 1) # obj[1]
sb2_x(obj, 1, red = TRUE) # obj[[1]]
sb2_x(obj, 1:2) # obj[1:2]
sb2_x(obj, is.numeric) # obj[sapply(obj, is.numeric)]
# for recursive subsets, see sb2_rec()


obj <- rbind(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_x(obj, 1:3, 1:ndims(obj))
# above is equivalent to obj[1:3, 1:3, drop = FALSE]
sb2_x(obj, i = is.numeric)
# above is equivalent to obj[sapply(obj, is.numeric)]
sb2_x(obj, c("a", "a"), 2L)
# above is equivalent to obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()]

obj <- array(as.list(1:64), c(4,4,3))
print(obj)
sb2_x(obj, n(1:3, 1:2), c(1,3))
# above is equivalent to obj[1:3, , 1:2, drop = FALSE]
sb2_x(obj, i = \(x)x > 5)
# above is equivalent to obj[sapply(obj, \(x) x > 5)]

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

# data.frame-like objects ====

obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb2_x(obj, 1:3, 1:3) # obj[1:3, 1:3, drop = FALSE]
sb2_x(obj, filter = ~ (a > 5) & (c < 19), vars = is.numeric)
```

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()` is a bit faster and uses less memory than apply.

Usage

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

Arguments

`x`	a mutable_atomic matrix. Arrays are not supported.
`MARGIN`	a single integer scalar, giving the subscript to apply the function over. 1 indicates rows, 2 indicates columns.
`FUN`	the function to be applied. The function must return a vector of the same type of `x`, and the appropriate length (i.e. length `ncol(x)` when `MARGIN == 1` or length `nrow(x)` when `MARGIN == 2`).

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)
```

slice *Efficient Sequence-based Subset Methods on (Long) Vectors*

Description

The `slice_` - methods are similar to the `sb_` - methods, except they don't require an indexing vector, and are designed for memory efficiency.

Usage

```
slice_x(x, ...)

## Default S3 method:
slice_x(
  x,
  from = NULL,
  to = NULL,
  by = 1L,
  ...,
  use.names = TRUE,
  sticky = getOption("squarebrackets.sticky", FALSE)
)

slice_rm(x, ...)

## Default S3 method:
slice_rm(
  x,
  from = NULL,
  to = NULL,
  by = 1L,
  ...,
  use.names = TRUE,
  sticky = getOption("squarebrackets.sticky", FALSE)
)

slice_set(x, ...)

## Default S3 method:
slice_set(x, from = NULL, to = NULL, by = 1L, inv = FALSE, ..., rp, tf)
```

Arguments

`x`	an atomic object. For `slice_set` it must be a mutable_atomic **variable**.

...	see squarebrackets_method_dispatch.
from, to, by	see cp_seq.
use.names	Boolean, indicating if flat names should be preserved. Note that, since slice operates on flat indices only, dimensions and dimnames are always dropped.
sticky	see squarebrackets_options.
inv	Boolean, indicating whether to invert the sequence. If TRUE, slice_set() will apply replacement/transformation on all elements of the vector, **except** for the elements of the specified sequence.
rp, tf	see squarebrackets_modify.

Value

Similar to the sb_ methods.

Examples

```
x <- mutable_atomic(1:1e7)

# extract:
slice_x(x, 1, 10)

# reverse:
slice_x(x, -1i, 1) |> head()

# remove:
slice_rm(x, 1, -11i) # all elements except the last 10

# replace every other element:
x <- mutable_atomic(1:1e7)
slice_set(x, 2, -1i, 2, rp = -1)
head(x)

# replace all elements except the first element:
x <- mutable_atomic(1:1e7)
slice_set(x, 1, 1, inv = TRUE, rp = -1)
head(x)
```

---

sub2ind *Convert Subscripts to Coordinates, Coordinates to Flat Indices, and Vice-Versa*

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; it performs a very simple (one might even say naive) conversion.
- `sub2ind()` is a faster and more memory efficient version of `coord2ind(sub2coord(sub, x.dims), x.dims)` (especially for up to 8 dimensions).

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

Value

For `sub2coord()` and `ind2coord()`:
Returns an integer matrix of coordinates (with properties as described in argument `coord`).

For `coord2ind()`:
Returns an numeric 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`)

For `sub2ind()`:
Returns an integer vector of flat indices(if `prod(x.dim) < (2^31 - 1)`), or an numeric vector of flat indices (if `prod(x.dim) >= (2^31 - 1)`).

Note

These functions were not specifically designed for duplicate indices per-sé.
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
```

`tci_bool` *Type Cast Indices*

Description

These functions typecast indices to proper integer indices.

Usage

```
tci_bool(indx, n, inv = FALSE, .abortcall = sys.call())

tci_int(indx, n, inv = FALSE, chkdup = FALSE, .abortcall = sys.call())

tci_chr(
  indx,
  nms,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)

tci_cplx(indx, n, inv = FALSE, chkdup = FALSE, .abortcall = sys.call())
```

Arguments

indx	the indices to typecast
n	the relevant size, when typecasting integer or logical indices. Examples: • If the target is row indices, input nrow for n. • If the target is flat indices, input the length for n.
inv	Boolean, indicating if the indices should be inverted. See squarebrackets_indx_args.
.abortcall	environment where the error message is passed to.
chkdup	see squarebrackets_options. for performance: set to FALSE
nms	the relevant names, when typecasting character indices. Examples: • If the target is row indices, input row names for nms. • If the target is flat indices, input flat names for nms.
uniquely_named	Boolean, indicating if the user knows a-priori that the relevant names of x are unique. If set to TRUE, speed may increase. But specifying TRUE when the relevant names are not unique will result in incorrect output.

Value

An integer vector of type-cast indices.

Examples

```
x <- matrix(1:25, 5, 5)
colnames(x) <- c("a", "a", "b", "c", "d")
print(x)

bool <- sample(c(TRUE, FALSE), 5, TRUE)
```

```
int <- 1:4
chr <- c("a", "a")
cplx <- 1:4 * -1i
tci_bool(bool, nrow(x))
tci_int(int, ncol(x), inv = TRUE)
tci_chr(chr, colnames(x))
tci_cplx(cplx, nrow(x))

ci_flat(x, 1:10 * -1i)
ci_margin(x, 1:4, 2)
ci_sub(x, n(1:5 * -1i, 1:4), 1:2)
```

Index

[.mutable_atomic (class_mutable_atomic), 41
[<-.mutable_atomic (class_mutable_atomic), 41

aaa00_squarebrackets_help, 3
aaa01_squarebrackets_supported_structures, 8
aaa02_squarebrackets_indx_fundamentals, 10
aaa03_squarebrackets_indx_args, 13
aaa04_squarebrackets_method_dispatch, 19
aaa05_squarebrackets_modify, 21
aaa06_squarebrackets_options, 24
aaa07_squarebrackets_PassByReference, 26
aaa08_squarebrackets_coercion, 29
aaa09_squarebrackets_inconveniences, 34
abind, *38*
ACCESS SUBSETS, *5*
ActiveBindings, *27*
all classes, *17*
apply, *83*
argument: chkdup, *24*
argument: sticky, *25*
array, *42*
arrayInd, *86*
as.mutable_atomic, *27*
as.mutable_atomic (class_mutable_atomic), 41
asub, *14*
atomic, *8*

bind, 36
bind_, *5*
bind_array (bind), 36
bind_dt (bind), 36
bind_mat (bind), 36
bindingIsLocked, *26*
BY, *54*

c, *6*, *63*
c.mutable_atomic (class_mutable_atomic), 41
cbind, *37, 38*
character, *42, 43*
ci_, *6*
ci_df (ci_flat), 39
ci_flat, 39
ci_margin (ci_flat), 39
ci_sub (ci_flat), 39
class: atomic array, *14*, *16*
class: atomic matrix, *14*, *16*
class: atomic vector, *14*
class: data.frame-like, *16, 17*
class: derived atomic vector, *14*
class: recursive array, *14*, *16*
class: recursive matrix, *14*, *16*
class: recursive vector, *14*
class_mutable_atomic, 41
coercion_by_reference: depends, *32*
coercion_by_reference: NO, *31, 32*
coercion_by_reference: YES, *32*
coercion_through_copy: depends, *30*
coercion_through_copy: NO, *31, 32*
coercion_through_copy: YES, *30–32*
complex, *11*, *42, 43*
coord2ind, *6*
coord2ind (sub2ind), 85
coord2sub (sub2ind), 85
couldb.mutable_atomic (class_mutable_atomic), 41
cp_seq, 44, *57*, *85*
currentBindings, *6*, *28*, 45

data.frame, *4*, *8*, *35*
data.table, *4*, *8*, *35*, *75*
DEVELOPER FUNCTIONS, *6*
double, *42, 43*
drop, *19*
dt, 48
dt_, *6*
dt_aggregate (dt), 48
dt_setadd (dt), 48
dt_setcoe, *31*
dt_setcoe (dt), 48

dt_setreorder (dt), 48
dt_setrm (dt), 48

EXTEND BEYOND, *5*
Extract, *43*

factor, *4*, *8*, *25*
fmatch, *61*
for performance: set to FALSE, *38*, *41*, *49*, *53*, *63*, *69*, *73*, *76*, *86*, *88*
format.mutable_atomic (class_mutable_atomic), 41
future_lapply, *23*

gsplit, *61*

HELPER FUNCTIONS, *6*

idx, *4*, *5*, *16*, *17*, *20*, *21*, *36*, 51, *68*
idx_by, *6*, 54
idx_ord_, *6*
idx_ord_df (idx_ord_v), 55
idx_ord_m (idx_ord_v), 55
idx_ord_v, 55
idx_r, *6*, *44*, 57
ind2coord (sub2ind), 85
indx_rm, *6*
indx_rm (indx_x), 58
indx_x, *6*, 58
integer, *42*, *43*
integer64, *42*, *43*
is.array, *9*
is.atomic, *9*
is.data.frame, *9*
is.matrix, *9*
is.mutable_atomic (class_mutable_atomic), 41
is.recursive, *9*

lapply, *23*
list, *6*, *63*
lockBinding, *28*, *46*
logical, *42*, *43*
lst, 58
lst_nlists, *59*
lst_nlists (lst), 58
lst_untree, *6*
lst_untree (lst), 58

ma_setv, *6*, 62
match_all, *6*, *11*, 61
MODIFY SUBSETS, *5*
mutable classes, *36*
mutable object, *5*
mutable_atomic, *4*, *6*, *8*, *9*, *25*, *27*, *32*, *36*, *38*, *63*, *75*, *83*, *84*
mutable_atomic (class_mutable_atomic), 41

n, *6*, *15*, 63
names, *43*
ndims, *6*, *9*, *15*, 64

option: squarebrackets.chkdup, *24*
option: squarebrackets.sticky, *25*
order, *49*, *55*, *56*

pass-by-reference, *9*
pass-by-reference semantics, *5*, *6*, *8*, *48*, *62*, *75*, *78*, *83*
print.mutable_atomic (class_mutable_atomic), 41

rapply, *59*
raw, *42*, *43*
rbind, *37*, *38*
recursive, *8*
rename a mutable object, *5*
rm, *47*
rrapply, *59*, *60*

sb2_mod, *5*, *17*, *22*
sb2_mod (sb_mod), 67
sb2_rec, *5*, *12*, *13*, *20*, *58*, 65
sb2_recin, *5*, *13*, *58*
sb2_recin (sb2_rec), 65
sb2_rm, *5*, *20*
sb2_rm (sb_rm), 72
sb2_set, *5*, *17*, *22*
sb2_set (sb_set), 75
sb2_setVarnames (sb_setRename), 78
sb2_x, *5*, *12*, *20*
sb2_x (sb_x), 80
sb_mod, *5*, *17*, *18*, *22*, *30–32*, *36*, 67
sb_rm, *5*, *18*, 72
sb_set, *5*, *6*, *17*, *18*, *22*, *31*, *32*, *36*, *42*, 75
sb_setDimnames (sb_setRename), 78
sb_setFlatnames (sb_setRename), 78
sb_setRename, 78
sb_x, *5*, *18*, *21*, *24*, 80
setapply, *6*, 83
setcolorder, *49*
setNames, *42*
setv, *62*
slice, *35*, *44*, 84
slice_rm, *5*, *25*, *26*
slice_rm (slice), 84

slice_set, *5*, *22*
slice_set (slice), 84
slice_x, *5*, *25*, *26*
slice_x (slice), 84
sort, *56*
SPECIALIZED FUNCTIONS, *5*
squarebrackets
(aaa00_squarebrackets_help), 3
squarebrackets-package
(aaa00_squarebrackets_help), 3
squarebrackets_coercion, *23*, *26*
squarebrackets_coercion
(aaa08_squarebrackets_coercion), 29
squarebrackets_help
(aaa00_squarebrackets_help), 3
squarebrackets_inconveniences
(aaa09_squarebrackets_inconveniences), 34
squarebrackets_indx_args, *35*, *41*, *49*, *53*, *54*, *58*, *69*, *73*, *76*, *81*, *88*
squarebrackets_indx_args
(aaa03_squarebrackets_indx_args), 13
squarebrackets_indx_fundamentals, *13–17*
squarebrackets_indx_fundamentals
(aaa02_squarebrackets_indx_fundamentals), 10
squarebrackets_method_dispatch, *5*, *9*, *53*, *65*, *69*, *73*, *76*, *79*, *81*, *85*
squarebrackets_method_dispatch
(aaa04_squarebrackets_method_dispatch), 19
squarebrackets_modify, *69*, *76*, *85*
squarebrackets_modify
(aaa05_squarebrackets_modify), 21
squarebrackets_options, *41*, *49*, *53*, *69*, *73*, *76*, *85*, *88*
squarebrackets_options
(aaa06_squarebrackets_options), 24
squarebrackets_PassByReference, *22*, *41*, *46*
squarebrackets_PassByReference
(aaa07_squarebrackets_PassByReference), 26
squarebrackets_supported_structures, *4*, *69*, *73*, *81*
squarebrackets_supported_structures
(aaa01_squarebrackets_supported_structures), 8
sticky, *26*
sub2coord, *6*
sub2coord (sub2ind), 85
sub2ind, 85
subscripts, *14*
supported mutable classes, *76*, *79*
supported mutable object, *75*, *78*

tci_, *6*
tci_bool, 87
tci_chr (tci_bool), 87
tci_cplx (tci_bool), 87
tci_int (tci_bool), 87
tibble, *35*
tidytable, *35*
typecast.mutable_atomic
(class_mutable_atomic), 41
typeof, *43*

view, *20*

which, *18*, *63*