Package 'squarebrackets'

November 13, 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 of 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,

2 Contents

```
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

aaa0_squarebrackets_help
aaa1_squarebrackets_supported_structures
aaa2_squarebrackets_coercion
aaa3_squarebrackets_indx_args
aaa4_squarebrackets_method_dispatch
aaa5_squarebrackets_modify
aaa6_squarebrackets_options
aaa7_squarebrackets_PassByReference
aaa8_squarebrackets_inconveniences
bind
ci_flat
class_mutable_atomic
cp_seq
currentBindings
dt 41
idx
idx_by
idx_ord_v
idx_r
indx_x
lst
match_all
ma_setv
n
ndims
sh2 rec

aaa0	SC	nuare	braci	kets	hel	n

sb_mod sb_rm sb_set sb_x setapply slice	
sb_set sb_x setapply slice	
sb_x	
setapply	
slice	
1.01.1	
sub2ind	
tci_bool	

aaa0_squarebrackets_help

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

3

82

Description

Index

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

Goal & Properties

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

But in some situations the square brackets operators are occasionally less than optimally convenient (see squarebrackets_inconveniences).

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.

These alternative sub-setting methods and functions 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 [<- 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.

Supported Classes

'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).

Supported immutable classes:

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

Supported mutable classes:

mutable_atomic, data.table (including tidytable, sf-data.table, and sf-tidytable).

Key-value storages, such as environments, and the various classes of the 'collections' package, are not supported.

Methods

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_.

The binding implementations for atomic dimensional objects (atomic arrays) start with bind_.

The binding implementations for recursive dimensional objects (recursive arrays, data.frames) start with bind2_.

There is also the somewhat separate idx method, which works on both recursive and non-recursive

objects.

And finally there are the slice_methods, which (currently) only work on atomic vectors.

ACCESSOR METHODS

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.

MODIFICATION METHODS

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 a (partial) copy of an object with modified (transformed or replaced) subsets.
- 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.

EXTENDING METHODS

Methods to extend or re-arrange an object beyond its current size:

- bind_, bind2_: 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.

So for example, use sb_rm() to remove subsets from atomic arrays, and use sb2_rm() to remove subsets from recursive arrays.

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 array, 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.

Author(s)

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

References

The badges shown in the documentation of this R-package were made using the services of: https://shields.io/

aaa1_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'). 'squabrackets 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).
- derived atomic vectors (factors, dates, date-time, etc.).
- 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 and bind_ implementations provided by 'squarebrackets' work on **atomic** (see is.atomic) objects.

The sb2_ methods and bind2_ implementations provided by 'squarebrackets' work on **recursive** (see is.recursive) objects.

The distinction is made between atomic and recursive methods, because R's S3 method dispatch only checks the class of an object.

So the S3 method dispatch makes no distinction, for example, between atomic matrices, and recursive matrices, despite them being completely different kinds of objects (except that they're both 2 dimensional objects).

See squarebrackets_method_dispatch for more details.

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).

'squarbrackets' 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 passby-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 'forcats' package for handling factors, and the 'anytime' package to handle datetime objects.

Not Supported S3 structures

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

aaa2_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 'forcats' package for handling factors, and the 'anytime' package to handle ddate-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))</pre>
```

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 \leftarrow 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).

${\bf Data. frames, when \ partially \ replacing/transforming \ columns}$

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 = \(x)x^2\)</pre>
```

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

```
)
# 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(</pre>
 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 = \sim (a >= 2) & (c <= 17), vars = is.numeric,
 # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
sb2_mod(
 obj, filter = \sim (a >= 2) & (c <= 17), vars = is.numeric,
 coe = as.double, tf = sqrt # SAFE: coercion performed
# sb_set():
obj <- data.table::data.table(</pre>
 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`)
 obj, filter = \sim (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(</pre>
 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 = \sim (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</pre>
```

aaa3_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.
- row, col: to specify rows and/or columns in tabular objects.
- sub, dims: to specify indices of arbitrary dimensions in arrays.
- filter, vars: to specify rows and/or columns specifically in data.frame-like objects.
- margin, slice: to specify indices of one particular dimension.

In this help page x refers to the object on which subset operations are performed.

Fundamentals

Base 'R' supports indexing through logical, integer, and character vectors.

'squarebrackets' suppports 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 $1:10 \times -1i$, 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.

What follows are detailed descriptions of the common arguments in 'squarebrackets' used to select indices.

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 strictly positive whole numbers vector with indices.
- a **complex** vector, as explained at the start of this help page.
- 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]
```

Arguments row, col

class: atomic matrix class: recursive matrix 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 strictly positive whole numbers vector with dimension indices to select for the operation.
- a complex vector, as explained at the start of this help page.
- a **logical** vector of the same length as the corresponding dimension size, giving the indices of this dimension to select for the operation.
- a character vector of index names.

If a dimension has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

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

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

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

Argument Pair sub, dims

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 of the same length as sub, giving the dimensions over which to select indices (i.e. dims specifies the "non-missing" index margins).

sub must be a list of subscripts, of the same length as dims. Each element of sub 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 strictly positive whole numbers vector with dimension indices to select for the operation.
- a **complex** vector, as explained at the start of this help page.
- a **logical** vector of the same length as the corresponding dimension size, giving the indices of this dimension to select for the operation.
- a character vector of index names.
 If a dimension has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.

Note also the following:

- If dims is a single integer, one can specify sub as an atomic vector of any of the above specified types, instead of as a list of length 1.
- As stated, dims specifies which index margins are non-missing.
 If dims and thus also sub is of length 0, it is taken as "all index margins are missing".

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

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

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

For a brief explanation of the relationship between flat indices (i), and dimensional subscripts (sub, dims), see the Details section in sub2ind.

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 Pair margin, slice

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;

The slice argument can be any of the following:

- a strictly positive whole numbers vector with dimension indices to select for the operation.
- a complex vector, as explained at the start of this help page.
- 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 result of a zero-length index specification depends on the rule-set of [.class(x)] and [<-.class(x)].

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 $sb_mod(x, col = 1:2, tf = tf)$ 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] \leftarrow tf(x[, -1:-2])
```

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</pre>
```

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

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

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

Regarding Performance

Integer indices and logical indices are the fastest.

Indexing through names or levels (i.e. character vectors) is the slowest.

Thus if performance is important, use integer or logical indices.

References

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

```
aaa4_squarebrackets_method_dispatch {\it Method\ Dispatch\ of\ 's quarebrackets'}}
```

Description

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

Atomic vs Recursive

With the exception of the idx method, the main generic methods are available in 2 forms: The Atomic form (sb_), and the Recursive Form (sb2_).

This because some S3 classes are available in both atomic and recursive forms.

For example, the array S3 class and the matrix S3 class (which inherits from the "array" S3 class) have both an atomic form, and a recursive form.

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.

- **copy semantics**: One can copy a subset of a recursive object without copying the rest of the object (thus saving memory when copying). For atomic objects, however, a copy operation 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 [[, [[<-)]. For atomic objects, these 2 have no meaningful difference (safe for perhaps attribute handling).
- **views**: For recursive objects, one can create a subset view of a recursive subset. Recursive subset views do not exist for atomic objects.

Despite these differences, R'S S3 method dispatch system does not have a built-in method to have separate dispatches for recursive and atomic objects.

Hence, given all the above, the 'squarebrackets' package has separate methods for recursive and atomic objects.

Manual Dispatch

The 'squarebrackets' package intentionally exports each function in its S3 method dispatch system. This is handy for programming purposes.

For example: atomic matrices and atomic arrays each have their own dispatch.

Thus, when looping though matrices and arrays to extract some elements, it may be easier to treat them all as arrays (remember that matrices inherit from arrays).

Thus one can use sb_x.array() to ensure the "array" method is used, instead of the "matrix" method.

Another advantage is that one can alias a specific dispatch of a method, if one so desires.

I.e.: array2_x <- sb2_x.array.

Under certain circumstances, this may 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).

aaa5_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</pre>
```

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 method to modify through copy.

This method always copies the modification.

For recursive objects, sb_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 method 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.

Arguments rp, tf, .lapply

Using the rp argument in the modification methods, corresponds to something like the following:

Using the tf argument (and .lapply argument, for recursive objects) in the modification methods, corresponds to something like the following:

```
x[...] \leftarrow tf(x[...]) \# for atomic objects 
 <math>x[...] \leftarrow .lapply(x[...], tf) \# for recursive objects
```

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

For recursive objects, tf 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.

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.

aaa6_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.

Mutable Atomic Messages

option: squarebrackets.ma_messages

The [<-.mutable_atomic method notifies the user of copy-on-modification.

Should the user find this annoying, the user can disable these messages by setting squarebrackets.ma_messages to FALSE.

aaa7_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)</pre>
```

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)</pre>
```

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' semanthics.

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)</pre>
```

Notice in the above code that mypointer has the same address as xa, and is therefore not a copy of xa.

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 current-Bindings function).

To prevent modification of an object's binding, 2 things must be true:

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

Some packages that provide pass-by-reference semantics tend to ignore the lock of an object's binding.

Use the 'squarebrackets' methods and (of course) core/base 'R' methods, in case the user fears the binding locks will not be respected.

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. 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!</pre>
```

```
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</pre>
```

aaa8_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</pre>
```

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.

When extracting/removing dimensional sub-set from an array, i.e. x[indx1, indx2], dimnames(x) will be preserved (within the range of the subset of course), but names(x) will be dropped completely.

The methods provided by 'squarebrackets' do not completely remove names(x) when extracting/removing subsets.

After all: why bother giving arrays flat names if they'll be stripped anyway?

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.

30 bind

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_ and bind2_ implementations provide dimensional binding functionalities. bind_ is for atomic objects, and bind2_ for recursive objects.

When possible, the bind_/bind2_ functions return mutable classes.

The following implementations are available:

- bind_mat() binds dimensionless atomic vectors and atomic matrices row- or column-wise. Returns a mutable_atomic matrix.
- bind2_mat() binds dimensionless recursive vectors and recursive matrices row- or columnwise.

Returns a recursive matrix (immutable).

- bind_array() binds atomic arrays and matrices. Returns a mutable_atomic array.
- bind2_array() binds recursive arrays and matrices. Returns a recursive array (immutable).

bind 31

bind2_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_"/"bind2_" 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 bind2_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, deparse.level = 1)
bind2_mat(arg.list, along, deparse.level = 1)
bind2_array(
    arg.list,
    along,
    name_along = TRUE,
    comnames_from = 1L,
    name_flat = FALSE
)
bind2_array(
    arg.list,
    along,
    name_along = TRUE,
    comnames_from = 1L,
    name_flat = FALSE
)
bind2_dt(arg.list, along)
```

Arguments

arg.list

a list of only the appropriate objects.

Do not mix recursive and atomic objects in the same list, as that may result in unexpected results.

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.

32 bind

deparse.level see cbind and rbind.

name_along Boolean, for bind_array() and bind2_array().

Indicates if dimension along should be named.

comnames_from integer scalar or NULL, for bind_array() and bind2_array().

Indicates which object in arg.list should be used for naming the shared di-

mension.

If NULL, no communal names will be given.

For example:

When binding columns of atomic matrices, comnames_from = 1 results in bind_array()

using rownames(arg.list[[1]]) for the row names of the output.

name_flat Boolean, for bind_array() and bind2_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() and bind2_array() are modified versions of the fantastic abind::abind function by Tony Plare and Richard Heiberger (see reference below).

bind_array() has slightly better performance than abind::abind, and has more streamlined naming options.

bind2_array() also has the streamlined naming options, and additionally differs from abind::abind in that it can handle recursive arrays properly (the original abind::abind function would unlist everything to atomic arrays).

bind_mat() and bind2_mat() are modified versions of rbind/cbind.

The primary differences is that bind_mat()/bin2_mat() give an error when fractional recycling is attempted (like binding 1:3 with 1:10).

Value

The new object.

References

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

Examples

```
# atomic arrays ====
x <- matrix(1:12,3,4)
dimnames(x) <- n(letters[1:3], LETTERS[1:4])
names(x) <- month.abb
print(x)
y <- x+100
arg.list <- list(x = x, y = y)
bind_array(arg.list, along=0) # binds on new dimension before first
bind_array(arg.list, along=1) # binds on first dimension
bind_array(arg.list, along=2)</pre>
```

ci_flat 33

```
bind_array(arg.list, along=3) # bind on new dimension after last
```

```
# recursive arrays ====
x <- matrix(as.list(1:12),3,4)
dimnames(x) <- n(letters[1:3], LETTERS[1:4])
names(x) <- month.abb
print(x)
y <- lapply(x, \(x) + 100)
dim(y) <- dim(x)
arg.list <- list(x = x, y=y)
bind2_array(arg.list, along=0) # binds on new dimension before first
bind2_array(arg.list, along=1) # binds on first dimension
bind2_array(arg.list, along=2)
bind2_array(arg.list, along=3) # bind on new dimension after last</pre>
```

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,
```

34 ci_flat

```
inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
ci_sub(
  х,
  sub,
  dims,
  inv = FALSE,
  chkdup = FALSE,
  uniquely_named = FALSE,
  .abortcall = sys.call()
)
ci_df(
  х,
  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 incor-

rect output.

.abortcall environment where the error message is passed to.

Value

An integer vector of casted indices.

Examples

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

class_mutable_atomic 35

```
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)</pre>
```

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.
- materialize_atomic(): takes an immaterial ALTREP atomic object, and returns a materialized mutable_atomic object.
- 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, ...)
is.mutable_atomic(x)
```

```
couldb.mutable_atomic(x)

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

materialize_atomic(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, ...)</pre>
```

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(), materialize_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.

cp_seq 37

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)</pre>
```

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 \boldsymbol{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 ${\bf maximally\ allowed}$ end value.
by	integer, of the same length as m or of length 1, specifying the step size.

38 *cp_seq*

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 whe 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\ {	t 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</pre>
```

currentBindings 39

```
)
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)]</pre>
```

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.

40 currentBindings

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)</pre>
```

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)</pre>
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:
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 \leftarrow 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"
 )
}
\label{eq:rm(list=c("x", "y", "z")) # clean up} \\
```

Functional Forms of data.table Operations

Description

dt

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 col-

umn 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 $x \leftarrow dt_setcoe(x, ...)$.

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)
}</pre>
```

```
# 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 = \sim (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 = \sim (a >= 2) & (c <= 17), vars = is.numeric,
 tf = sqrt # SAFE: coercion performed; so no warnings
)
str(obj)
# dt_setrm ====
obj <- data.table::data.table(</pre>
 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(</pre>
 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(</pre>
 a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
new <- data.table::data.table(</pre>
 e = sample(c(TRUE, FALSE), 10, TRUE),
  f = sample(c(TRUE, FALSE), 10, TRUE)
dt_setadd(obj, new)
print(obj)
```

idx 45

```
# 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)</pre>
```

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_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</pre>
```

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').

46 idx

Usage

```
idx(x, ...)
## Default S3 method:
idx(x, i, inv = FALSE, ..., chkdup = getOption("squarebrackets.chkdup", FALSE))
## S3 method for class 'array'
idx(
  х,
  sub = NULL,
  dims = NULL,
  slice = NULL,
  margin = NULL,
  i = NULL,
  inv = FALSE,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
## S3 method for class 'data.frame'
idx(
  Х,
  slice,
  margin,
  inv = FALSE,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
```

Arguments

```
vector, matrix, array, or data.frame; both atomic and recursive objects are supported.
see squarebrackets_method_dispatch.
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.
```

 idx_by 47

Examples

```
# atomic ====
x <- 1:10
x[idx(x, (x)x>5)] < -5
print(x)
x \leftarrow 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 \leftarrow 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(</pre>
 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.

 idx_by

Usage

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

Arguments

```
the object from which to compute the indices.
Х
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
                  a factor giving the groups.
grp
parallel, mc.cores
                  see BY.
```

Value

A vector of indices.

Examples

```
# vectors ====
(a <- 1:20)
(grp <- factor(rep(letters[1:5], each = 4)))</pre>
# get the last element of `a` for each group in `grp`:
i <- idx_by(a, 0L, last, grp)</pre>
sb_x(cbind(a, grp), row = i)
# data.frame ====
x <- data.frame(</pre>
  a = sample(1:20),
  b = letters[1:20],
  group = factor(rep(letters[1:5], each = 4))
print(x)
\mbox{\tt\#} get the first row for each group in data.frame \mbox{\tt`x\':}
row <- idx_by(x, 1, first, x$group)</pre>
sb2_x(x, row)
\# get the first row for each group for which a > 10:
x2 <- sb2_x(x, filter = ~a > 10)
row <- na.omit(idx_by(x2, 1, first, x2$group))</pre>
sb2_x(x2, row)
```

idx_ord_v 49

idx_ord_v

Compute Ordered Indices

Description

Computes ordered indices. Similar to order, except the user must supply a vector, a list of equallength 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(
 х,
 na.last = TRUE,
 decr = FALSE,
 method = c("auto", "shell", "radix")
idx_ord_m(
 Х,
 margin,
 na.last = TRUE,
 decr = FALSE,
 method = c("auto", "shell", "radix")
)
idx_ord_df(
 х,
 na.last = TRUE,
 decr = FALSE,
```

 idx_r

```
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))</pre>
```

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.
```

indx_x 51

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)]</pre>
```

 $indx_x$

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.

52 lst

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))</pre>
```

lst

Unnest Tree-like List to Recursive 2d Array or Flattened Recursive Vector

Description

[[, [[<-, sb2_rec, and sb2_recin, can performing recursive subset operation 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 2d recursive array (i.e. a list-matrix), allowing vectorized subset operations to be performed on the list.

1st_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 = FALSE)
```

Arguments

Х

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 2D recursive array (i.e. a matrix of lists), 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 2D recursive array (i.e. a matrix of lists), with length(x) columns and n rows, where n = sapply(x, lst_nlists) |> max().
 Empty elements will be filled with list(NULL).

lst 53

use.names

Boolean, indicating if the elements returned from lst_untree() should be named.

Names of nested elements, such as x[[c("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.

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.

(One advantage of a recursive matrix over a data.frame, is that a recursive matrix can have separate column names and regular names, whereas the names of a data.frame are necessarily equal to the column names).

For lst_nlists():

A single integer, giving the total number of recursive list-elements in the given list.

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 = 2, use.names = TRUE)</pre>
dim(y)
print(y)
sb2_x(y, 1:3, 1:2) # vectorized selection of multiple recursive elements
# un-tree row-wise:
```

54 match_all

```
sapply(x, lst_nlists) |> max() # number of columns `y` will have
y <- lst_untree(x, margin = 1, use.names = TRUE)</pre>
dim(y)
print(y)
sb2_x(y,1:2, 1:3) # vectorized selection of multiple recursive elements
# simple flattened list:
y <- lst_untree(x, margin = 0, use.names = TRUE)</pre>
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 \leftarrow lapply(1:10, \x) list(sample(letters), sample(1:10)))
sapply(x, lst_nlists) |> max()
y <- lst_untree(x, margin = 1)</pre>
dim(y)
print(y)
lst_untree(x, margin = 1)
# showcasing vectorized sub-setting ====
x \leftarrow 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)</pre>
# getting the first recursive elements in the second level/depth in base R:
for(i in seq_along(x)) {
 x[[c(i, c(1,1))]]
# the same, but vectorized using the untree'd list:
y[seq_len(nrow(y)), 1]
```

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 55

```
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)</pre>
```

56 ma_setv

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 \boldsymbol{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 $x \leftarrow ma_setv(x, ...)$.

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

n 57

Examples

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

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]</pre>
```

ndims

Get Number of Dimensions

Description

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

Usage

ndims(x)

Arguments

Х

the object to get the number of dimensions from.

sb2_rec

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)</pre>
```

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.

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, rec)
sb2_recin(x, rec, rp, tf)
```

sb2_rec 59

Arguments

x a list, or list-like object.

rec a strictly positive integer vector or character vector, of length p, such that 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 index-

ing by name, due to the recursive nature of this type of sub-setting.

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.

- When specifying rp = NULL, will **remove** (recursive) subset x[[rec]]. To specify actual NULL instead of removing a subset, use 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.

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

tf

Since recursive objects are pointers to 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(</pre>
```

60 sb2_rec

```
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")</pre>
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 $sb_{mod}(x, ...)$ 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(
  х,
  i = NULL,
  inv = FALSE,
  . . . ,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
## S3 method for class 'matrix'
sb_mod(
  Х,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  . . . ,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
## S3 method for class 'array'
sb_mod(
  х,
  sub = NULL,
  dims = NULL,
  i = NULL,
  inv = FALSE,
  rp,
```

```
tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
sb2\_mod(x, ...)
## Default S3 method:
sb2_mod(
  х,
  i = NULL,
  inv = FALSE,
  . . . ,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)
## S3 method for class 'matrix'
sb2_mod(
  х,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  . . . ,
  rp,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
## S3 method for class 'array'
sb2_mod(
  х,
  sub = NULL,
  dims = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)
## S3 method for class 'data.frame'
sb2_mod(
 Х,
 row = NULL,
  col = NULL,
  filter = NULL,
```

```
vars = NULL,
inv = FALSE,
coe = FALSE,
...,
rp,
tf,
chkdup = getOption("squarebrackets.chkdup", FALSE),
.lapply = lapply
)
```

Arguments

```
see squarebrackets_supported_structures.
Х
                  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
                  Either FALSE (default), TRUE, or a function.
coe
                  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 \leftarrow matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")</pre>
print(obj)
rp <- -1:-9
sb_{mod}(obj, 1:3, 1:3, rp = rp)
# above is equivalent to obj[1:3, 1:3] <- -1:-9; obj
sb_{mod}(obj, i = (x)x \le 5, rp = -1:-5)
# above is equivalent to obj[obj <= 5] <- -1:-5; obj</pre>
sb_{mod}(obj, col = "a", rp = -1:-8)
# above is equivalent to obj[, which(colnames(obj) %in% "a")] <- -1:-8; obj
sb_{mod}(obj, 1:3, 1:3, tf = \(x) -x)
# above is equivalent to obj[1:3, 1:3] \leftarrow (-1 * obj[1:3, 1:3]); obj
sb_mod(obj, i = \(x)x \le 5, tf = \(x) -x)
# above is equivalent to obj[obj \le 5] <- (-1 * obj[obj \le 5]); obj
obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")</pre>
print(obj)
sb_{mod}(obj, 1:3, 1:3, tf = (x) -x)
# above is equivalent to obj[1:3, 1:3] \leftarrow -1 * obj[1:3, 1:3]
sb_mod(obj, i = \(x)x \le 5, tf = \(x) -x)
# above is equivalent to obj[obj \le 5] < -1:-5; obj
sb_mod(obj, col = "a", tf = \(x) -x)
# above is equivalent to obj[, which(colnames(obj) %in% "a")] <- -1:-8; obj
obj <- array(1:64, c(4,4,3))
print(obj)
sb_{mod}(obj, list(1:3, 1:2), c(1,3), rp = -1:-24)
# above is equivalent to obj[1:3, , 1:2] <- -1:-24
sb_{mod}(obj, i = (x)x \le 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</pre>
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</pre>
# dimensional lists ====
obi <- 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")</pre>
print(obj)
sb2\_mod(obj, 1:3, 1:3, 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)</pre>
sb2\_mod(obj, col = c("a"), 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(obi)
sb2_mod(obj, list(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 = \sim (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 = \sim (a >= 2) & (c <= 17), vars = is.numeric,
```

66 sb_rm

```
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
)</pre>
```

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 sb_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 'matrix'
sb_rm(
  х,
  row = NULL,
  col = NULL,
  i = NULL,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
## S3 method for class 'array'
sb_rm(
  Х,
  sub = NULL,
  dims = NULL,
  i = NULL,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
sb2_rm(x, ...)
## Default S3 method:
```

sb_rm 67

```
sb2_rm(
     Х,
      i = NULL
      red = FALSE,
      chkdup = getOption("squarebrackets.chkdup", FALSE)
   ## S3 method for class 'matrix'
    sb2_rm(
      х,
     row = NULL,
      col = NULL,
      i = NULL,
      red = FALSE,
      chkdup = getOption("squarebrackets.chkdup", FALSE)
   ## S3 method for class 'array'
    sb2_rm(
      Х,
      sub = NULL,
      dims = NULL,
      i = NULL,
      red = FALSE,
      chkdup = getOption("squarebrackets.chkdup", FALSE)
   ## S3 method for class 'data.frame'
    sb2_rm(
      х,
      row = NULL,
      col = NULL,
      filter = NULL,
      vars = NULL,
      chkdup = getOption("squarebrackets.chkdup", FALSE)
    )
Arguments
                    see squarebrackets_supported_structures.
   Х
                    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

68 sb_rm

red Boolean, for list only.

I f red = TRUE, selecting a single element with non-empty arguments 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) \leftarrow c("a", "b", "c", "a")
print(obj)
sb_rm(obj, 1:3, 1:3)
# above is equivalent to obj[-1:-3, -1:-3, drop = FALSE]
sb_rm(obj, i = (x)x>5)
# above is equivalent to obj[!obj > 5]
sb_rm(obj, col = "a")
# above is equivalent to obj[, which(!colnames(obj) %in% "a")]
obj <- array(1:64, c(4,4,3))
print(obj)
sb_rm(obj, n(1, c(1, 3)), c(1, 3))
# 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()</pre>
```

sb_set 69

```
# dimensional lists ====
obj <- rbind(
  lapply(1:4, \x) sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \x) sample(1:10)),
  lapply(1:4, \xspace (x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
colnames(obj) <- c("a", "b", "c", "a")</pre>
print(obj)
sb2_rm(obj, 1:3, 1:3)
# 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, col = c("a", "a"))
# 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) \times 5)]
# data.frame-like objects ====
obj \leftarrow 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-byreference 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, ...)
```

sb_set

```
## Default S3 method:
sb_set(
 Х,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
## S3 method for class 'matrix'
sb_set(
  х,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
## S3 method for class 'array'
sb_set(
  Х,
  sub = NULL,
  dims = NULL,
  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(
  Х,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  inv = FALSE,
  ...,
```

sb_set 71

```
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 $x \leftarrow sb_set(x, ...)$. Since this function returns void, you'll just get NULL.

Examples

```
# mutable_atomic objects ====
gen_mat <- function() {
  obj <- as.mutable_atomic(matrix(1:16, ncol = 4))
  colnames(obj) <- c("a", "b", "c", "a")
  return(obj)
}
obj <- obj2 <- gen_mat()
obj
sb_set(obj, 1:3, 1:3, rp = -1:-9)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, i = \(x\)x<=5, rp = -1:-5)</pre>
```

```
obj2
obj <- obj2 <- gen_mat()</pre>
sb_set(obj, col = "a", rp = cbind(-1:-4, -5:-8))
obj <- obj2 <- gen_mat()</pre>
obj
sb_set(obj, 1:3, 1:3, tf = \(x) -x)
obi2
obj <- obj2 <- gen_mat()</pre>
sb_set(obj, i = \(x)x \le 5, tf = \(x) -x)
obj2
obj <- obj2 <- gen_mat()</pre>
obi
sb_set(obj, col = "a", tf = \(x) -x)
obj2
gen_array <- function() {</pre>
 as.mutable_atomic(array(1:64, c(4,4,3)))
obj <- gen_array()</pre>
obj
sb_set(obj, list(1:3, 1:2, c(1, 3)), 1:3, rp = -1:-12)
obj <- gen_array()</pre>
obj
sb_set(obj, i = (x)x <= 5, rp = -1:-5)
# 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`)
 obj, filter = \sim (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 = \sim (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(
```

sb_x 73

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

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 sb_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 'matrix'
sb_x(x, row = NULL, col = NULL, i = NULL, ...)
## S3 method for class 'array'
sb_x(x, sub = NULL, dims = NULL, i = NULL, ...)
sb2_x(x, ...)
## Default S3 method:
sb2_x(x, i = NULL, red = FALSE, ...)
## S3 method for class 'matrix'
sb2_x(x, row = NULL, col = NULL, i = NULL, red = FALSE, ...)
## S3 method for class 'array'
sb2_x(x, sub = NULL, dims = NULL, 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 lists only, indicating if the result should be reduced.

If red = TRUE, selecting a single element with non-empty arguments 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")</pre>
print(obj)
sb_x(obj, 1:3, 1:3)
# above is equivalent to obj[1:3, 1:3, drop = FALSE]
sb_x(obj, i = (x)x>5)
# above is equivalent to obj[obj > 5]
sb_x(obj, col = c("a", "a"))
# above is equivalent to obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()]
obj <- array(1:64, c(4,4,3))
print(obj)
sb_x(obj, n(1:3, 1:2), c(1,3))
# 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()
# dimensional lists ====
obj <- rbind(</pre>
 lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
 lapply(1:4, \(x)sample(1:10)),
 lapply(1:4, \(x)rnorm(10)),
```

setapply 75

```
lapply(1:4, \(x)sample(letters))
colnames(obj) <- c("a", "b", "c", "a")</pre>
print(obj)
sb2_x(obj, 1:3, 1:3)
# 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, col = c("a", "a"))
# 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) \times 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.

76 slice

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)</pre>
```

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)
slice_rm(x, ...)
```

slice 77

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

Arguments

an atomic object. Х For slice_set it must be a mutable_atomic variable. see squarebrackets_method_dispatch. from, to, by see cp_seq. Boolean, indicating if flat names should be preserved. use.names Note that, since slice operates on flat indices only, dimensions and dimnames are always dropped. Boolean, indicating whether to invert the sequence. inv 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)</pre>
```

78 sub2ind

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.

sub2ind 79

x. dim an integer vector giving the dimensions of the array in question. I.e. dim(x).

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.

Details

coord

Subscripts and coordinates only exist for dimensional objects (such as arrays).

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.

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 by 4 matrix, is as follows:

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

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

The functions described here thus follow also this convention.

Value

For sub2coord() and ind2coord():

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

For coord2ind():

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

For coord2sub():

80 tci_bool

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

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())
```

tci_bool 81

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 \boldsymbol{x} are

unique.

If set to TRUE, speed may increase.

But specifying TRUE when the relevant names are not unique will result in incor-

rect output.

Value

An integer vector of casted 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)</pre>
```

Index

[.mutable_atomic	c.mutable_atomic
(class_mutable_atomic), 35	<pre>(class_mutable_atomic), 35</pre>
[<mutable_atomic< td=""><td>cbind, <i>32</i></td></mutable_atomic<>	cbind, <i>32</i>
(class_mutable_atomic), 35	character, <i>35, 36</i>
	ci_, 6
aaa0_squarebrackets_help, 3	ci_df(ci_flat), 33
aaa1_squarebrackets_supported_structures,	ci_flat, 33
7	ci_margin(ci_flat),33
aaa2_squarebrackets_coercion, 9	ci_sub(ci_flat), 33
aaa3_squarebrackets_indx_args, 14	class: atomic array, 17, 18
aaa4_squarebrackets_method_dispatch,	class: atomic matrix, 16
20	class: atomic vector, 16
aaa5_squarebrackets_modify, 21	class: data.frame-like, 16-18
aaa6_squarebrackets_options, 23	class: derived atomic vector, 16
aaa7_squarebrackets_PassByReference,	class: recursive array, 17, 18
24	class: recursive matrix, 16
aaa8_squarebrackets_inconveniences, 28	class: recursive vector, 16
abind, 32	class_mutable_atomic, 35
ACCESSOR METHODS, 5	coercion_by_reference: depends, 12
ActiveBindings, 25	coercion_by_reference: NO, 11
all classes, 18	coercion_by_reference: YES, 11
apply, 75	coercion_through_copy: depends, 9, 10
• • • •	coercion_through_copy: NO, 10, 11
argument: chkdup, 24	coercion_through_copy: YES, 9-11
array, 20, 36	complex, 15, 35, 36
arrayInd, 78	coord2ind, 6
as.mutable_atomic, 26	coord2ind (sub2ind), 78
as.mutable_atomic	coord2sub (sub2ind), 78
(class_mutable_atomic), 35	couldb.mutable_atomic
asub, 17	<pre>(class_mutable_atomic), 35</pre>
atomic, 7	cp_seq, 37, 50, 77
	currentBindings, 6, 27, 39
bind, 30	
bind2_, 5	data.frame, 7, 29
bind2_array (bind), 30	data.table, 7, 29, 69
bind2_dt (bind), 30	DEVELOPER FUNCTIONS, 6
bind2_mat (bind), 30	double, <i>35</i> , <i>36</i>
bind_, 5	drop, 20
bind_array (bind), 30	dt, 41
bind_mat(bind), 30	dt_, 5
bindingIsLocked, 25	dt_aggregate (dt), 41
BY, 48	dt_setadd(dt), 41
	dt_setcoe, 11
c, 6, 57	dt_setcoe(dt), 41
	• • •

INDEX 83

dt_setreorder (dt), 41	mutable classes, 30
dt_setrm(dt), 41	mutable object, 5
	mutable_atomic, 4, 6-8, 12, 25, 26, 30, 56,
EXTENDING METHODS, 5	69, 75, 77
Extract, 36	<pre>mutable_atomic(class_mutable_atomic),</pre>
	35
fmatch, <i>55</i>	33
for performance: set to FALSE, 32, 34,	n, <i>6</i> , <i>17</i> , <i>57</i>
43, 46, 56, 63, 67, 71, 79, 81	names, 36
<pre>format.mutable_atomic</pre>	ndims, 6, 8, 57
<pre>(class_mutable_atomic), 35</pre>	11011113, 0, 0, 37
future_lapply, 23	option: squarebrackets.chkdup, 24
gsplit, 55	option: squarebrackets.ma_messages, 24
	order, 43, 49, 50
HELPER FUNCTIONS, 6	
	pass-by-reference, 8, 9
idx, 4, 5, 18, 20, 22, 30, 45, 61	pass-by-reference semantics, $5-7$, 42 , 56 ,
idx_by, 6, 47	69, 75
idx_ord_, 6	<pre>print.mutable_atomic</pre>
idx_ord_df (idx_ord_v), 49	<pre>(class_mutable_atomic), 35</pre>
idx_ord_m(idx_ord_v), 49	
idx_ord_v, 49	rapply, 53
idx_r, 6, 37, 38, 50	raw, $35, 36$
ind2coord (sub2ind), 78	rbind, <i>32</i>
	recursive, 7
indx_rm, 6	rm, <i>40</i>
indx_rm (indx_x), 51	rrapply, 53
indx_x, 6, 51	
integer, 35, 36	sb2_mod, <i>5</i> , <i>18</i>
integer64, 35, 36	sb2_mod (sb_mod), 61
is.array, 8	sb2_rec, 5, 52, 58
is.atomic, 8	sb2_recin, 5, 52
is.data.frame, 8	sb2_recin(sb2_rec), 58
is.matrix, 8	sb2_rm, 5
is.mutable_atomic	
(class_mutable_atomic), 35	sb2_rm (sb_rm), 66
is.recursive, 8	sb2_set, 5, 18
	sb2_set (sb_set), 69
lapply, 23	sb2_x, 5
list, 6, 57	sb2_x (sb_x), 73
lockBinding, 27, 39, 40	sb_mod, 5, 10, 11, 18, 19, 22, 30, 61
logical, <i>35</i> , <i>36</i>	sb_rm, 5, 19, 66
1st, 52	sb_set, 5, 6, 10, 11, 18, 19, 22, 30, 35, 69
lst_nlists, 52	sb_x, 5, 19, 21, 24, 73
lst_nlists(lst), 52	setapply, $6,75$
lst_untree, 5	setcolorder, 43
lst_untree (lst), 52	setNames, 36
155_46. 55 (155), 62	setv, <i>56</i>
ma_setv, 6, 56	slice, 29, 37, 38, 76
match_all, 6, 15, 54	slice_rm, 5
materialize_atomic	slice_rm(slice), 76
(class_mutable_atomic), 35	slice_set, 5
matrix, 20	slice_set(slice), 76
MODIFICATION METHODS, 5	slice_x,5

84 INDEX

```
slice_x (slice), 76
                                                tci_bool, 80
sort. 50
                                                tci_chr(tci_bool), 80
SPECIALIZED FUNCTIONS, 5
                                                tci_cplx(tci_bool), 80
squarebrackets
                                                tci_int(tci_bool), 80
        (aaa0_squarebrackets_help), 3
                                                tibble, 29
squarebrackets-package
                                                tidytable, 29
        (aaa0_squarebrackets_help), 3
                                                typecast.mutable_atomic
squarebrackets_coercion, 23, 25
                                                        (class_mutable_atomic), 35
squarebrackets_coercion
                                                typeof, 36
        (aaa2_squarebrackets_coercion),
                                                view, 21
squarebrackets_help
                                                which, 19, 56
        (aaa0_squarebrackets_help), 3
squarebrackets_inconveniences, 3
squarebrackets_inconveniences
        (aaa8_squarebrackets_inconveniences),
squarebrackets_indx_args, 29, 34, 42, 46,
        47, 51, 63, 67, 71, 73, 81
squarebrackets_indx_args
        (aaa3_squarebrackets_indx_args),
squarebrackets_method_dispatch, 5, 8, 46,
        63, 67, 71, 73, 77
squarebrackets_method_dispatch
        (aaa4_squarebrackets_method_dispatch),
        20
squarebrackets_modify, 63, 71, 77
squarebrackets_modify
        (aaa5_squarebrackets_modify),
squarebrackets_options, 34, 43, 46, 63, 67,
        71,81
squarebrackets_options
        (aaa6_squarebrackets_options),
squarebrackets_PassByReference, 22, 35,
        39
squarebrackets_PassByReference
        (aaa7_squarebrackets_PassByReference),
squarebrackets_supported_structures,
        63, 67, 73
squarebrackets_supported_structures
        (aaa1_squarebrackets_supported_structures),
sub2coord, 6
sub2coord (sub2ind), 78
sub2ind, 17, 78
supported mutable classes, 71
supported mutable object, 69
tci_, 6
```