Package ‘squarebrackets’

September 8, 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, tibbles, data.tables, tibbles, 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 requir-
ing 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 2D recursive array of lists,
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.

License MIT + file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Suggests rlang,
knitr,
rmarkdown,

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

Depends R (>= 4.2.0)

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

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

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

Language en-gb

Contents

`aaa0_squarebrackets_help` *squarebrackets: Subset Methods as Alternatives to the Square Brackets Operators for Programming*

Description

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

Goal & Properties

Among programming languages, 'R' has perhaps one of the most flexible and comprehensive sub-setting functionality, provided by the square brackets operators (`[`, `[<-`).
But in some situations the square brackets operators are occasionally less than optimally convenient (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 copy of the object.
  - For sub-set operations that just change values in-place (similar to the `[<-` and `[[<-` methods) the user can choose a method that modifies the object by **reference**, or choose a method that returns a **deep copy**.
- **Careful handling of names**:
  - 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 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`).

There are, of course, a lot of classes which are not supported by 'squarebrackets'.
Most notably, key-value stores - such as environments, and the various 'collections' classes from the 'collections' package - are not supported.

Methods and Functions

GENERIC 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 non-recursive 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.

The available generic methods are the following:

- sb_x, sb2_x: extract, exchange, or duplicate subsets.
- sb_rm, sb2_rm: un-select/remove subsets.
- sb_set, sb2_set: modify (transform or replace) subsets of a mutable object using pass-by-reference semantics.
- sb_mod, sb2_mod: return a **copy** of an object with modified (transformed or replaced) subsets.
- sb2_rec: access recursive subsets of lists.
- sb2_reccom: replace, transform, remove, or add recursive subsets to a list, through R's default Copy-On-Modify semantics.
- sb_setRename, sb2_setRename: change the names of a mutable object using pass-by-reference semantics.
- bind_, bind2_: implementations for binding dimensional objects.
- idx: translate given indices/subscripts, for the purpose of copy-on-modify substitution.

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.

SPECIALIZED FUNCTIONS

Additional specialized sub-setting functions are provided:

- lst_untree: unnest tree-like nested list, to make vectorized sub-setting on recursive subsets of the list easier.
- 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.
- sb_str: extract or replace a subset of characters of a single string (each single character is treated as a single element).
- sb_a: extract multiple attributes from an object.

HELPER FUNCTIONS

A couple of 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.
- 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.

- Computing sequences:
  seq_rec2 for the recursive sequence generator (for example to generate a Fibonacci sequence).

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_immutable_classes`
Supported Immutable S3 Classes, With Auto-Coercion Rules

Description

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

- base `atomic` vector classes
  (atomic vectors, matrices, and arrays).
- classes derived from `atomic` vectors
  (factors, date, POSIXct, etc.).
  'squarebrackets' treats these classes as regular atomic vectors.
- base list classes
  (recursive vectors, matrices, and arrays)
  (note that lists are merely pointers to other objects, and these other objects may be of a different class and may even be mutable).
- data.frame
  (including the classes `tibble`, `sf-data.frame` and `sf-tibble`).

Note that "immutable" does not mean you cannot modify it.
It simply means that modification leads to a copy being made.

Auto-Coercion Rules

Atomic
coercion_through_copy: YES
Atomic objects are automatically coerced to fit the modified subset values, when modifying through copy.
For example, replacing one or multiple values in an integer vector (type `int`) with a decimal number (type `dbl`) will coerce the entire vector to type `dbl`.

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 ddatetime objects.

List
coercion_through_copy: depends
Lists themselves allow complete change of their elements, since lists are merely pointers.
For example, the following code performs full coercion:

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

However, a recursive subset of a list which itself is not a list, follows the coercion rules of whatever class the recursive subset is.
For example the following code:

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

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

Data.frames when replacing/transforming whole columns
coercion_through_copy: YES
A data.frame is actually a list, where each column is itself a list. As such, replacing/transforming whole columns, so `row = NULL` and `filter = NULL`, allows completely changing the type of the column.
Note that coercion of columns needs arguments `row = NULL` and `filter = NULL` in the sb_mod and

sb_set methods; no auto-coercion will take place when specifying something like `row = 1:nrow(x)` (see next section).

Data.frames, when partially replacing/transforming columns
coercion_through_copy: NO
If rows are specified in the sb_mod and sb_set methods, and thus not whole columns but parts of columns are replaced or transformed, no auto-coercion takes place.
I.e.: replacing/transforming a value in an integer (`int`) column to become `1.5`, will not coerce the column to the decimal type (`dbl`); instead, the replacement value `1.5` is coerced to integer `1`.
The `coe` argument in the sb_mod method allows the user to enforce coercion, even if subsets of columns are replaced/transformed instead of whole columns.
Specifically, the `coe` arguments allows the user to specify a coercive function to be applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected.
This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`).

Examples

```
# Coercion examples - lists ====
x <- list(factor(letters), factor(letters))
print(x)
sb2_mod(x, 1, rp = list(1)) # first element fully changed.

x <- list(1:10, 1:10)
print(x)
sb2_reccom(x, 1, rp = "a") # coerces first element to character
print(x)


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


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

obj <- data.frame(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

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


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

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

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

---

aaa2_squarebrackets_mutable_classes
Supported Mutable S3 Classes, With Auto-Coercion Rules

Description

The sb_ generic methods support the following Mutable S3 classes:

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

The mutable version of the `list` class would be the environment class, and the various key-value storage classes available in other packages, such as the 'collapse' package.
Key-value storage classes generally do not use square brackets for their primary sub-setting method, and are thus not covered by this package.

Auto-Coercion Rules

Coercion Semantics
The mutable classes support "copy-on-modify" semantics like the immutable classes, but - unlike the immutable classes - they also support "pass-by-reference" semantics.
The sb_mod method modify subsets of an object through a **deep copy**.
The sb_set method and dt_setcoe function modify subsets of an object **by reference**.
These 2 copy semantics - "pass by reference" or "modify copy" - have slightly different auto-coercion rules.
These are explained in this section.

mutable_atomic
coercion_through_copy: YES
coercion_by_reference: NO
Mutable atomic objects are automatically coerced to fit the modified subset values, when modifying

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

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

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

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

data.table, when replacing/transforming whole columns
coercion_through_copy: YES
coercion_by_reference: YES
A data.table is actually a list made mutable, where each column is itself a list. As such, replacing/transforming whole columns, so `row = NULL` and `filter = NULL`, allows completely changing the type of the column.
Note that coercion of columns needs arguments `row = NULL` and `filter = NULL` in the sb_mod and sb_set methods; no auto-coercion will take place when specifying something like `row = 1:nrow(x)` (see next section).

data.table, when partially replacing/transforming columns
coercion_through_copy: NO
coercion_by_reference: NO
If rows are specified in the sb_mod and sb_set methods, and thus not whole columns but parts of columns are replaced or transformed, no auto-coercion takes place.
I.e.: replacing/transforming a value in an integer (`int`) column to become `1.5`, will not coerce the column to the decimal type (`dbl`); instead, the replacement value `1.5` is coerced to integer `1`.
The `coe` argument in the sb_mod method allows the user to enforce coercion, even if subsets of columns are replaced/transformed instead of whole columns.
Specifically, the `coe` arguments allows the user to specify a coercive function to be applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected.
This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`).

Views of Lists
coercion_by_reference: depends
Regular lists themselves are not treated as mutable objects by 'squarebrackets'.
However, lists are not actually really objects, merely a (potentially hierarchical) structure of pointers.
Thus, even if a list itself is not treated as mutable, subsets of a list which are themselves mutable classes, are mutable.
For example, if you have a list of `data.table` objects, the data.tables themselves are mutable.
Therefore, the following will work:

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

Notice in the above code that `mypointer` is not a copy of `x$a`, since they have the same address.
Thus changing `mypointer` also changes `x$a`.
In other words: `mypointer` is what could be called a "view" of `x$a`.
Notice also that `sb_set(x$a, ...)` will not work, since `sb_set()` requires **actual variables**, similar to in-place functions in the style of `` `myfun()<-` ``.

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

Examples

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

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

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

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

# sb_mod():
obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_mod(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)

# sb_set():
sb2_set(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)
str(obj)

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


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

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

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

# sb_set():
obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt
  # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)

obj <- data.table::data.table(
  a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10])
)
str(obj)
dt_setcoe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb2_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by dt_setcoe(); so no warnings
)
print(obj)


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

# View of List ====

x <- list(
 a = data.table::data.table(cola = 1:10, colb = letters[1:10]),
 b = data.table::data.table(cola = 11:20, colb = letters[11:20])
)
print(x)
mypointer <- x$a
address(mypointer) == address(x$a) # they are the same
sb2_set(mypointer, col = "cola", tf = \(x)x^2)
print(x) # notice x has been changed
```

`aaa3_squarebrackets_indx_args`
Index Arguments in the Generic Sub-setting Methods

Description

There are 6 types of arguments that can be used in the generic methods of 'squarebrackets' to specify the indices to perform operations on:

- `i`: to specify flat (i.e. dimensionless) indices.
- `row, col`: to specify rows and/or columns in tabular objects.
- `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.

Integers
Integers are the most basic form on index selection.
All forms of indexing in 'squarebrackets' are internally translated to integer indexing first, ensuring consistency.
Indexing through integers 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.

Complex Numbers
Unlike base 'R', squarebrackets' includes support for indexing through `complex` vectors.

Indexing with complex numbers is a generalization of indexing with regular integers.

It works as follows:
A complex vector is essentially structured as `Real + Imaginary`.
The Real part must be **strictly positive integers**.
The Imaginary part must be **constant**.

- If `Imaginary` is positive or zero, it works the same as integer values.
- If `Imaginary` is negative, indexing counts backwards, where the integer indices are computed as `n - Real + 1L`.

Where `n` is the maximum possible integer (i.e. `length(x)`, or `dim(x)[L]`, depending on the situation).

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: factor
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 integer** 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 integer** 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 integer** 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;
etc.

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

- a **strictly positive integer** 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] <- tf(x[, -1:-2])
x
```

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

All NULL indices

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

```
x[]
```

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

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

The above is true **even if** `inv = TRUE` and/or `drop = 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 `NA`s are ignored.

Disallowed Combinations of Index Arguments

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

One cannot specify `row` and `filter` simultaneously; it's either one or the other.
One cannot specify `col` and `vars` simultaneously; it's either one or the other.
One cannot specify the `sub, dims` pair and `slice, margin` pair simultaneously; it's either one pair or the other pair.
In the above cases it holds that if one set is specified, the other is set is ignored.

Drop

Sub-setting with the generic methods from the 'squarebrackets' R-package using dimensional arguments (`row, col, lyr, sub, dims, filter, vars`) always use `drop = FALSE`.
To drop potentially redundant (i.e. single level) dimensions, use the drop function, like so:

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

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_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`.

squarebrackets.protected

The user should NEVER touch the `squarebrackets.protected` option.
This option lists all locked non-functions in the base environment, in order to protect them from any accidental pass-by-reference modification by the methods/functions from 'squarebrackets'.
Other packages that provide pass-by-reference modification, such as the 'collapse' package, generally do not provide such protections, and are not blocked by `squarebrackets.protected`.

`aaa5_squarebrackets_method_dispatch`
Method Dispatch of 'squarebrackets'

Description

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

Atomic vs Recursive

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.
The recursive form of arrays and matrices is sometimes referred to as a "dimensional list".

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

- **homo- or heterogeneous**: atomic object are homogeneous, in that they can only contain one data-type (logical, integer, double, character, complex, raw).
  In contract, recursive objects are heterogeneous, as they can have any combination of data-types.

- **vectorization**: vectorized operations generally work on atomic objects, whereas recursive objects generally require loops or apply-like functions.
  This is especially relevant for transforming subsets.
- **recursive subsets**: Recursive objects distinguish between "regular" subset operations (in base R using `[`, `[<-`), and recursive subset operations (in base R using `[[`, `[[<-`).
  For both forms, atomic objects give atomic objects.
  But for recursive objects, these 2 subset operations are significantly different.
- **views**: Recursive objects are weird in that they are *pointers* to other objects.
  As such they allow their extracted subsets to be *views* of these pointers (see squarebrackets_mutable_classes for more information on how to use "views" of recursive objects).
  Atomic objects do now allow for subset views.

The 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).

`aaa6_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.
But it does have several disadvantages.
First, the coercion rules are slightly different: see squarebrackets_mutable_classes.

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

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

modifies not just `x`, but also `y`.
This is true even if one of the variables is locked (see bindingIsLocked).
I.e. the following code,

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

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

Mutable vs Immutable Classes

With the exception of environments, most of base R's data types are treated as immutable:
Modifying an object in 'R' will make a copy of the object, something called 'copy-on-modify' semantics.
However, almost any of base R's data types can be modified by reference, through R's own 'C' API, or through 'C++' code (like via 'Rcpp'), thus treating these objects as mutable, even though they are not "supposed" to be mutable.
Modifying a base 'R' object by reference can be problematic.
Since 'R', and also most R-packages, treat these objects as immutable, modifying them as-if they are mutable may produce undesired results.
For example, modifying `base::leters` by reference will actually modify the `letters` object within `base`.

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

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

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.

A more subtle type of immaterial objects are ALTREP objects.
ALTREP objects store instructions on how values are stored, but do not actually store the values.
For example, `x <- 1:1e6` is an **immaterial** object:
Unlike `rnorm(1e6)`, `1:1e6` does not actually store 1 million values;
Rather, it stores the simple **instruction** that `x[i] = i`.
When `x` is modified, the given instructions obviously don't hold any more, and so 'R' will materialize `x`, which means `x` will then actually store its values in memory.
So when `x` is materialized, the size of `x` in the memory will change from a few bytes to a few Mega Bytes.

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 **do not work** on immaterial objects.

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.
This works since a `data.table` is a recursive object.
Pass-by-Reference never works on atomic ALTREP objects.

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, than 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:

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

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

Input Variable

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

Lock Binding

Mutable classes are, as the name suggests, meant to be mutable.
Locking the binding of a mutable object is **mostly** fruitless (but not completely; see the currentBindings function).
To 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.

Protected Addresses

To prevent an accidental pass-by-reference modification of objects in the base environment, all addresses of all exported objects in the base environment (baseenv) are stored in the option `squarebrackets.protected` whenever 'squarebrackets' is **loaded**, either directly or indirectly.
Needless to say, the user should never touch this option.

Protection

Due to the properties described above in this help page, something like the following will not work:

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

The above won't work 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.

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

Some packages provide functions that change class-related attributes of objects by reference.
Using such functions is discouraged, unless you know exactly what you're doing.

Examples

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

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

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

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

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

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

---

aaa7_squarebrackets_inconveniences

Examples Where the Square Bracket Operators Are Less Convenient

---

Description

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

Arrays

In order to perform subset operations on some array `x` with the square brackets operator (`[`, `[<-`), one needs to know how many dimensions it has.
I.e. if x has 3 dimensions, one would use:

```
x[i, j, k, drop = FALSE]

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

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

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

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.
This attempt to keep data.frame-like classes consistent does, admittedly, result in some oddities in how data.frames are treated by 'squarebrackets', compared to how other classes are treated by 'squarebrackets':

- Whole-columns will be auto-coerced when replaced/transformed by sb_mod, but partial columns will not be auto-coerced by default.
- The sb_x and sb_rm methods always automatically conserve all attributes (though names and dimensions are adjusted accordingly, of course);
  the attributes are not stripped, unlike the other classes.
- Giving a data.frame-like object with non-unique column names to the `sb_`-methods returns an error;
  duplicating columns with sb_x will automatically adjust the column names to make them unique.

Annoying Sub-setting By Names

When selecting names for sub-setting, only the first occurrences of the names are selected for the sub-set;
and when un-selecting/removing names for sub-setting, the syntax is very different from selecting names.

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

Modification Semantics

'R' adheres to copy-on-modify semantics when replacing values using `[<-`.
But sometimes one would like explicit control when to create a copy, and when to modify using pass-by-reference semantics.

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

Regarding Other Packages

There are some packages that solve some of these issues.
But using different packages for solving different issues for the same common theme (in this case: solving some inconveniences in the square bracket operators) leads to inconsistent code.
I have not found an R-package that provides a holistic approach to providing alternative methods to the square brackets operators.
Thus, this 'R' package was born.

`bind` *Dimensional Binding of Objects*

Description

The `bind_` 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_array()` binds atomic arrays and matrices.
  Returns a mutable_atomic array.
- `bind2_array()` binds recursive arrays and matrices.
  Returns dimensional lists.
- `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.

Usage

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

bind2_array(
  arg.list,
  along,
  name_along = TRUE,
  name_shared = 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.
`name_along`	Boolean, for `bind_array()` and `bind2_array()`. Indicates if dimension `along` should be named.
`name_shared`	integer or `NULL`, for `bind_array()` and `bind2_array()`. Indicates which object in `arg.list` should be used for naming the shared dimension. If `NULL`, no shared names will be given. For example: When binding columns of atomic matrices, `name_shared = 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).

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, name_flat = TRUE) # binds on new dimension before first
bind_array(arg.list, along=1, name_flat = TRUE) # binds on first dimension
bind_array(arg.list, along=2, name_flat = TRUE)
bind_array(arg.list, along=3, name_flat = TRUE) # bind on new dimension after last


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


# recursiv 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, name_flat = TRUE) # binds on new dimension before first
bind2_array(arg.list, along=1, name_flat = TRUE) # binds on first dimension
bind2_array(arg.list, along=2, name_flat = TRUE)
bind2_array(arg.list, along=3, name_flat = TRUE) # bind on new dimension after last
```

`ci_flat` *Construct Indices*

Description

These functions construct flat or dimensional indices.

- `ci_flat()` constructs an integer vector flat indices.
- `ci_margin()` constructs an integer vector of indices for one particular dimension margin.

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

Usage

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

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

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

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

Arguments

`x`	the object for which the indices are meant.

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

Value

An integer vector of casted indices.

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
```

`class_mutable_atomic` *Mutable Atomic Classes*

Description

The `mutable_atomic` class is a mutable version of atomic classes.
It works exactly the same in all aspects as regular atomic classes, with only one real difference:
The 'squarebrackets' methods and functions that perform modification by reference (basically all methods and functions with "set" in the name) accept `mutable_atomic`, but do not accept regular `atomic`.
See squarebrackets_PassByReference for details.

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

Exposed functions (beside the S3 methods):

- `mutable_atomic()`: create a `mutable_atomic` object.
- `is.mutable_atomic()`: checks if an object is atomic.
- `as.mutable_atomic()`: converts a regular atomic object to `mutable_atomic`.
- `couldb.mutable_atomic()`: checks if an object could be `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.

Usage

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

as.mutable_atomic(x, ...)

is.mutable_atomic(x)

couldb.mutable_atomic(x)

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

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

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

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

Arguments

`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.
`value`	see Extract.

Value

For `mutable_atomic()`:
Returns a `mutable_atomic` object.

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

For `is.mutable_atomic()`:

Returns `TRUE` if the object is atomic, has the class `"mutable_atomic"`, has the correctly set attribute `"typeof"`, **and** has an address that does not overlap with the addresses of base objects.
`is.mutable_atomic()` returns `FALSE` otherwise.

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

Warning

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

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

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

Description

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

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

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

See also squarebrackets_PassByReference for information regarding the relation between locked bindings and pass-by-reference modifications.

Usage

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

Arguments

`x`	the existing variable whose address to use when searching for bindings.
`action`	a single string, giving the action to perform. Must be one of the following: • `"list"` (default). • `"checklock"`. • `"lockbindings"`.
`env`	the environment where to look for objects. If `NULL` (default), the caller environment is used.

Details

The lockBinding function locks a binding of an object, preventing modification.
'R' also uses locked bindings to prevent modification of objects from package namespaces.
The pass-by-reference semantics of 'squarebrackets' in principle respect this, and disallows modification of objects by reference.

However, lockBinding does not lock the address/pointer of an object, only one particular binding of an object.
This problematic; consider the following example:

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

In the above code, `x` and `y` share the same address, thus pointing to the same memory, yet only `y` is actually locked.
Since `x` is not locked, modifying `x` is allowed.
But since `sb_set()`/`sb2_set()` performs modification by reference, `y` will still be modified, despite being locked.

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

Value

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

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

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

Warning

The `currentBindings()` function only locks **currently existing** bindings in the **caller environment**;
bindings that are created **after** calling `currentBindings()` will not automatically be locked.
Thus, every time the user creates a new binding of the same object, and the user wishes it to be locked, `currentBindings()` must be called again.

Examples

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


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


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

if(requireNamespace("tinytest")) {
  tinytest::expect_error(
    sb_set(x, i = 1, rp = -1),
    pattern = "object is locked"
  )
}
```

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

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


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

dt — *Functional Forms of data.table Operations*

Description

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

- `dt_aggregate()` aggregates a data.table or tidytable, and returns the aggregated copy.
- `dt_setcoe()` coercively transforms columns of a data.table or tidytable using pass-by-reference semantics.
- `dt_setrm()` removes columns of a data.table or tidytable using pass-by-reference semantics.
- `dt_setadd(x, new)` adds the columns from data.table/tidytable `new` to data.table/tidytable `x`, thereby modifying `x` using pass-by-reference semantics.
- `dt_setreorder()` reorders the rows and/or variables of a `data.table` using pass-by-reference semantics.

Usage

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

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

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

dt_setadd(x, new)

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

Arguments

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

Details

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

Value

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

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

Examples

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

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

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

  head(d.aggr)
}



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


# dt_setcoe ====

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
str(obj)
obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
dt_setcoe(obj, vars = is.numeric, v = as.numeric) # integers are now numeric
str(obj)
sb2_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed; so no warnings
)
str(obj)


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


# dt_setrm ====

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

```
dt_setrm(obj, col = 1)
str(obj)

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


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


# dt_setadd ====

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



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


# dt_setreorder====

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

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

Description

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

- `idx(x, i = i, ...)` converts linear indices to a strictly positive integer vector of linear indices.
- `idx(x, sub = sub, dims = dims, ...)` converts dimensional indices to a strictly positive integer vector of linear indices.

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

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

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

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

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

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

The `idx()` method is particularly handy for replacing or coercively transforming shallow subsets of recursive objects, without having to return a copy of the entire object.
Thus combining `[<-` with `idx()` is more efficient than sb2_mod for recursive objects.

Usage

```
idx(x, ...)

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

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

## S3 method for class 'data.frame'
idx(
  x,
  slice,
  margin,
  inv = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)
```

Arguments

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

Value

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

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

Examples

```
# atomic ====

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

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


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


# recursive ====

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

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

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

---

idx_by *Compute Grouped Indices*

Description

Given:

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

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

Usage

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

Arguments

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

Value

A vector of indices.

Examples

```

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

# get the last element of `a` for each group in `grp`:
i <- idx_by(a, 0L, last, grp)
sb_x(cbind(a, grp), row = i)


# data.frame ====
x <- data.frame(
  a = sample(1:20),
  b = letters[1:20],
  group = factor(rep(letters[1:5], each = 4))
)
print(x)
# get the first row for each group in data.frame `x`:
row <- idx_by(x, 1, first, x$group)
sb2_x(x, row)
# get the first row for each group for which a > 10:
x2 <- sb2_x(x, filter = ~ a > 10)
row <- na.omit(idx_by(x2, 1, first, x2$group))
sb2_x(x2, row)
```

`idx_ord_v` *Compute Ordered Indices*

Description

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

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

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

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

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

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

Usage

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

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

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

Arguments

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

Value

See order.

Examples

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

`idx_r` *Compute Integer Index Range*

Description

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

Usage

```
idx_r(x, m = 0L, start = NULL, end = NULL, by = 1L)
```

Arguments

`x`	the object to index.
`m`	integer, giving the margin. For non-dimensional objects or for flat indices, specify `m = 0L`.
`start`	integer or complex, of the same length as `m` or of length 1, specifying the start point.
`end`	integer or complex, , of the same length as `m` or of length 1, specifying the end point.
`by`	integer scalar, of the same length as `m` or of length 1, specifying the step size.

Details

start, stop, by
If `start, stop` 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).

If `start, stop, 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.

So, for example, to specify something like `n:1`, one can simply call `idx_r(x, m, by = -1L)`.

Heterogeneous Complex input
The user can specify complex vectors with varying imaginary parts for `start` and `end`.
Except for this difference, the complex vectors are translated to integer vectors as explained in squarebrackets_indx_args.

Multiple indices at once
The `idx_r()` function can compute indices for multiple dimensions at once, by specifying a vector for `m`.

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:(n-1)
sb2_x(x, ind1) # extract the row range

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

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

`indx_x` *Exported Utilities*

Description

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

Usage

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

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

Arguments

`i`	See squarebrackets_indx_args.
`x`	a vector, vector-like object, factor, data.frame, data.frame-like object, or a list.
`xnames`	names or dimension names
`xsize`	length or dimension size

Value

The subsetted object.

Examples

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

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

Description

`[[`, `[[<-`, sb2_rec, and sb2_reccom, 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.
`lst_untree()` can also simply flatten the list, making it a non-nested list.
See the Examples section to understand how the list will be arranged and named.

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

Usage

```
lst_nlists(x)

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

Arguments

`x`	a tree-like nested list.
`margin`	a single integer, indicating how the result should be arranged: • `margin = 0` produces a simple flattened recursive vector (i.e. list) without dimensions. • `margin = 1` produces a 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)`.

`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)
dim(y)
```

```
print(y)
y[["Y.Z.Y"]] # you can still use names for selecting/replacing
sb2_x(y, 1:3, 1:2) # vectorized selection of multiple recursive elements


# un-tree row-wise:
sapply(x, lst_nlists) |> max() # number of columns `y` will have
y <- lst_untree(x, margin = 1, use.names = TRUE)
dim(y)
print(y)
y[["Y.Z.Y"]] # you can still use names for selecting/replacing
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)
print(y)
y[["Y.Z.Y"]]
x[[c("Y", "Z", "Y")]] # equivalent in the original list


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

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

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

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

lst_untree(x, margin = 1)


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

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

# getting the first recursive elements in the second level/depth in base R:
for(i in seq_along(x)) {
  x[[c(i, c(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.

It is essentially a much more efficient version of:

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

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

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

Usage

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

Arguments

`needles, haystack`	vectors
`unlist`	Boolean, indicating if the result should be a single integer vector (`TRUE`, default), or a list (length = `length(needles)`) of integer vectors (`FALSE`).

Value

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

Examples

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

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

Description

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

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

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

Usage

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

Arguments

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

Value

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

Examples

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

`n` *Nest*

Description

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

Usage

```
n()
```

Value

The list.

Examples

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

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

Description

The `sb2_rec()` and `sb2_reccom()` methods are essentially convenient wrappers around `[[` and `[[<-`, respectively.

`sb2_rec()` will access recursive subsets of lists.

`sb2_reccom()` 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_reccom(x, rec, rp, tf)
```

Arguments

`x`	a list, or list-like object.
`rec`	an integer (including negative integers) 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 indexing by name, due to the recursive nature of this type of subsetting.
`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_reccom()` 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_reccom()` will add value `rp` to the list as a new element at the end.
`tf`	an optional function. If specified, performs `x[[rec]] <- tf(x[[rec]])`, using R's default Copy-On-Modify semantics. Does not support extending a list like argument `rp`.

Details

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

For `sb2_reccom(..., tf = tf)`:
Returns VOID, but transforms the specified recursive subset, using R's default Copy-On-Modify semantics.

Examples

```
lst <- list(
  A = list(
    A = list(A = "AAA", B = "AAB"),
    A = list(A  = "AA2A", B = "AA2B"),
    B = list(A = "ABA", B = "ABB")
  ),
  B = list(
    A = list(A = "BAA", B = "BAB"),
    B = list(A = "BBA", B = "BBB")
  )
)

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

# access recursive subsets ====

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


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

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

# replace "AAB" using R's default in-place semantics:
sb2_reccom(
  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_reccom(lst, i, rp = "AND THEN THERE WERE NONE")

print(lst)


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

# Modify View of List By Reference ====

x <- list(
 a = data.table::data.table(cola = 1:10, colb = letters[1:10]),
 b = data.table::data.table(cola = 11:20, colb = letters[11:20])
)
print(x)
mypointer <- sb2_rec(x, "a")
address(mypointer) == address(x$a) # they are the same
sb2_set(mypointer, col = "cola", tf = \(x)x^2)
print(x) # notice x has been changed
```

sb_mod *Method to Return a Copy of an Object With Modified Subsets*

Description

This is an S3 Method to return a copy of an object with modified subsets.
Use `sb_mod(x, ...)` if x is an atomic object.
Use `sb2_mod(x, ...)` if x is a recursive object (i.e. list or data.frame-like).

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

Usage

```
sb_mod(x, ...)

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

## S3 method for class 'matrix'
sb_mod(
  x,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

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

sb2_mod(x, ...)

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

## S3 method for class 'matrix'
sb2_mod(
  x,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)

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

```
sb2_mod(
  x,
  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(
  x,
  row = NULL,
  col = NULL,
  filter = NULL,
  vars = NULL,
  inv = FALSE,
  coe = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE),
  .lapply = lapply
)
```

Arguments

`x`	see squarebrackets_immutable_classes and squarebrackets_mutable_classes.
`...`	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`	an object of somewhat the same type as the selected subset of `x`, and the same same length as the selected subset of `x` or a length of 1. To remove recursive subsets of recursive objects, see either sb2_rec or sb2_rm.
`tf`	the transformation function.
`chkdup`	see squarebrackets_options. for performance: set to FALSE
`.lapply`	the generic methods use lapply for list- and data.frame-like objects to compute `tf()` on every list element or dataset column. 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.

`coe`	Either `FALSE` (default), `TRUE`, or a function. The argument `coe` is ignored if both the `row` and `filter` arguments are set to `NULL`. See Details section for more info. for performance: set to FALSE

Details

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

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

1. The user can supply a coercion function to argument `coe`.
   The function is applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected.
   This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`).
2. The user can set `coe = TRUE`.
   In this case, the whole columns specified in `col` or `vars` are extracted and copied to a list.
   Subsets of each list element, corresponding to the selected rows, are modified with `rp` or `tf()`, using R's regular auto-coercion rules.
   The modified list is then returned to the data.frame-like object, replacing the original columns.

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

Value

A copy of the object with replaced/transformed values.

Examples

```
# atomic objects ====

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

```
sb_mod(obj, 1:3, 1:3, tf = \(x) -x)
# above is equivalent to  obj[1:3, 1:3] <- (-1 * obj[1:3, 1:3]); obj
sb_mod(obj, i = \(x)x<=5, tf = \(x) -x)
# above is equivalent to  obj[obj <= 5] <- (-1 * obj[obj <= 5]); obj

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

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

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


# lists ====

obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb2_mod(obj, "a", rp = list(1L))
# above is equivalent to  obj[["a"]] <- 1L; obj
sb2_mod(obj, is.numeric, rp = list(-1:-10, -11:-20))
# above is equivalent to  obj[which(sapply(obj, is.numeric))] <- list(-1:-10, -11:-20); obj

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


# dimensional lists ====
obj <- rbind(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_mod(obj, 1:3, 1:3, 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)
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(obj)
```

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

`sb_rm` *Method to Un-Select/Remove Subsets of an Object*

Description

This is an S3 Method to un-select/remove subsets from an object.
Use `sb_rm(x, ...)` if `x` is an atomic object.
Use `sb2_rm(x, ...)` if `x` is a recursive object (i.e. list or data.frame-like).

Usage

```
sb_rm(x, ...)

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

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

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

sb2_rm(x, ...)

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

## S3 method for class 'matrix'
sb2_rm(
  x,
  row = NULL,
  col = NULL,
  i = NULL,
  drop = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

## S3 method for class 'array'
sb2_rm(
  x,
  sub = NULL,
  dims = NULL,
  i = NULL,
  drop = FALSE,
  ...,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

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

Arguments

`x`	see squarebrackets_immutable_classes and squarebrackets_mutable_classes.
`...`	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
`drop`	Boolean, for list only. I f `drop = TRUE`, selecting a single element with non-empty arguments will give the simplified result, like using `[[]]`. If `drop = FALSE`, a list is always returned regardless of the number of elements.

Value

A copy of the sub-setted object.

Examples

```
# atomic objects ====

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_rm(obj, 1:3, 1:3)
# 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, drop = TRUE)
# above is equivalent to obj[[!sapply(obj, is.numeric)]] IF this returns a single element
obj <- list(a = 1:10, b = letters[1:11], c = letters)
sb2_rm(obj, is.numeric)
# above is equivalent to obj[!sapply(obj, is.numeric)] # this time singular brackets?
# for recusive indexing, see sb2_rec()


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


# dimensional lists ====

obj <- rbind(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_rm(obj, 1:3, 1:3)
# 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) x > 5)]
```

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

# data.frame-like objects ====

obj <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
print(obj)
sb2_rm(obj, 1:3, 1:3)
# above is equivalent to obj[-1:-3, -1:-3, drop = FALSE]
sb2_rm(obj, filter = ~ (a > 5) & (c < 19), vars = is.numeric)
```

---

sb_set *Method to Modify Subsets of a Mutable Object By Reference*

Description

This is an S3 Method to replace or transform a subset of a supported mutable object using pass-by-reference semantics
Use `sb_set(x, ...)` if `x` is an atomic object (i.e. mutable_atomic).
Use `sb2_set(x, ...)` if `x` is a recursive object (i.e. data.table).

Usage

```
sb_set(x, ...)

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

## S3 method for class 'matrix'
sb_set(
  x,
  row = NULL,
  col = NULL,
  i = NULL,
  inv = FALSE,
  ...,
  rp,
  tf,
  chkdup = getOption("squarebrackets.chkdup", FALSE)
)

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

Arguments

`x`	a **variable** belonging to one of the supported mutable classes.
`...`	see squarebrackets_method_dispatch.
`i`, `row`, `col`, `sub`, `dims`, `filter`, `vars`, `inv`	See squarebrackets_indx_args. An empty index selection leaves the original object unchanged.
`rp`	an object of somewhat the same type as the selected subset of `x`, and the same same length as the selected subset of `x` or a length of 1. To remove recursive subsets of recursive objects, see either sb2_rec or sb2_rm.
`tf`	the transformation function.
`chkdup`	see squarebrackets_options. for performance: set to FALSE

`.lapply` the generic methods use lapply for list- and data.frame-like objects to compute `tf()` on every list element or dataset column.
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.

Details

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

Value

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

Examples

```
# mutable_atomic objects ====

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

obj <- obj2 <- gen_mat()
obj
sb_set(obj, 1:3, 1:3, rp = -1:-9)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, i = \(x)x<=5, rp = -1:-5)
obj2
obj <- obj2 <- gen_mat()
obj
sb_set(obj, col = "a", rp = cbind(-1:-4, -5:-8))
obj2

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

```
obj <- obj2 <- gen_mat()
obj
sb_set(obj, col = "a", tf = \(x) -x)
obj2


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


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

# data.table ====

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_set(
  obj, filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # WARNING: sqrt() results in `dbl`, but columns are `int`, so decimals lost
)
print(obj)

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
dt_setcoe(obj, vars = is.numeric, v = as.numeric)
str(obj)
sb2_set(obj,
  filter = ~ (a >= 2) & (c <= 17), vars = is.numeric,
  tf = sqrt # SAFE: coercion performed by dt_setcoe(); so no warnings
)
print(obj)

obj <- data.table::data.table(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
str(obj) # notice that columns "a" and "c" are INTEGER (`int`)
sb2_set(
  obj, vars = is.numeric,
  tf = sqrt # SAFE: row=NULL & filter = NULL, so coercion performed
)
str(obj)
```

sb_setRename *Method to Change the Names of a Mutable Object By Reference*

Description

This is an S3 Method to rename a supported mutable object using pass-by-reference semantics.

This method takes extra care not to modify any objects that happen to share the same address as the (dim)names of `x`.
I.e. the following code:

```
x <- mutable_atomic(1:26)
names(x) <- base::letters
y <- x
sb_setRename(x, newnames = rev(names(x)))
```

will not modify `base::letters`, even though `names(x)` shared the same address.
Thus, `sb_setRename()` can be used safely without fearing such accidents.

Use `sb_setRename(x, ...)` if `x` is a non-recursive object (i.e. mutable_atomic).
Use `sb2_setRename(x, ...)` if `x` is a recursive object (i.e. data.table).

Usage

```
sb_setRename(x, ...)

## Default S3 method:
sb_setRename(x, newnames, ...)

## S3 method for class 'array'
sb_setRename(x, newdimnames, newnames, ...)

sb2_setRename(x, ...)

## S3 method for class 'data.table'
sb2_setRename(x, old, new, skip_absent = FALSE, ...)
```

Arguments

`x`	a **variable** belonging to one of the supported mutable classes.
`...`	see squarebrackets_method_dispatch.
`newnames`	atomic character vector giving the new names. Specifying `NULL` will remove the names.
`newdimnames`	a list of the same length as `dim(x)`. The first element of the list corresponds to the first dimension (i.e. rows), the second element to the second dimension (i.e. columns), and so on. The components of the list can be either `NULL`, or a character vector with the same length as the corresponding dimension. Instead of a list, simply `NULL` can be specified, which will remove the `dimnames` completely.
`old`	the old column names
`new`	the new column names, in the same order as `old`
`skip_absent`	Skip items in old that are missing (i.e. absent) in `names(x)`. Default `FALSE` halts with error if any are missing.

Value

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

Examples

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

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


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


x <- mutable_atomic(
   1:20, letters[1:20], dim = c(5, 4), dimnames = n(letters[1:5], letters[1:4])
)
print(x)
sb_setRename(
  x,
  newdimnames = lapply(dimnames(x), rev),
  newnames = rev(names(x))
)
print(x)


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



# data.table ====

x <- data.table::data.table(
  a = 1:20,
  b = letters[1:20]
)
print(x)
sb2_setRename(x, old = names(x), new = rev(names(x)))
print(x)
```

`sb_special` *Specialized Sub-setting Functions*

Description

The `sb_a()` function subsets extracts one or more attributes from an object.

The `sb_str()` function subsets characters of single string, or replace a subset of the characters of a single string with the subsets of the characters of another string. In both cases, a single string is treated as a iterable vector, where each single character in a string is a single element. The `sb_str()` function is considerably faster than doing the equivalent operation in base 'R' or even 'stringi'.

Usage

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

sb_a(x, a = NULL)
```

Arguments

`str`	a single string.
`ind`	an integer vector, giving the positions of the string to subset.

`rp.str`, `rp.ind`	similar to `str` and `ind`, respectively.
	If not specified, `sb_str()` will perform something like
	`str[ind]`
	treating `str` as an iterable vector.
	If these ARE specified, `sb_str()` will perform something like
	`str[ind] <- rp.str[rp.ind]`
	treating `str` and `rp.str` as iterable vectors.

`x`	an object
`a`	a character vector of attribute names. If `NULL` (default), ALL attributes are returned.

Value

The sub-setted object.

Examples

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


x <- "hello"
sb_str(x, 5:1) # this gives "olleh"
sb_str(x, c(1:5, 5)) # this gives "helloo"
sb_str(x, c(2:5)) # this gives "ello"
sb_str(x, seq(1, 5, by = 2)) # this gives "hlo"
sb_str(x, 1:4, "world", 1:4) # this gives "worlo"
```

---

sb_x *Method to Extract, Exchange, or Duplicate Subsets of an Object*

Description

This is an S3 Method to extract, exchange, or duplicate (i.e. repeat x times) subsets of an object.
Use `sb_x(x, ...)` if x is an atomic object.
Use `sb2_x(x, ...)` if x is a recursive object (i.e. list or data.frame-like).

Usage

```
sb_x(x, ...)

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

## S3 method for class '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, drop = FALSE, ...)

## S3 method for class 'matrix'
sb2_x(x, row = NULL, col = NULL, i = NULL, drop = FALSE, ...)

## S3 method for class 'array'
sb2_x(x, sub = NULL, dims = NULL, i = NULL, drop = FALSE, ...)

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

Arguments

`x`	see squarebrackets_immutable_classes and squarebrackets_mutable_classes.
`...`	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.
`drop`	Boolean, for lists only. If `drop = TRUE`, selecting a single element with non-empty arguments will give the simplified result, like using `[[]]`. If `drop = FALSE`, a list is always returned regardless of the number of elements.

Value

Returns a copy of the sub-setted object.

Examples

```
# atomic objects ====

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_x(obj, 1:3, 1:3)
# 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, drop = 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(
  lapply(1:4, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:4, \(x)sample(1:10)),
  lapply(1:4, \(x)rnorm(10)),
  lapply(1:4, \(x)sample(letters))
)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb2_x(obj, 1:3, 1:3)
# 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) x > 5)]

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

# data.frame-like objects ====

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

---

seq_rec2 *Generate Recursive Sequence Through Repeated Arithmetic Infix Operations*

Description

This is a recursive sequence generator.
The function is essentially a highly generalized version of a Fibonacci sequence generator.

Starting with 2 initial values, each next value `i` is generated by either one of 2 formulas:

1. `x[i] = (s[1] + m[1] * x[i-1]) %inop% (s[2] + m[2] * x[i-2])`
2. `x[i] = (m[1] * (x[i-1] + s[1])) %inop% (m[2] * (x[i-2] + s[2]))`

where `%inop%` is the arithmetic infix operator chosen,
and `m` and `s` are each a numeric vector of length 2.
The order of `x[i-1]` and `x[i-2]` can also be swapped.

Usage

```
seq_rec2(
  inits = c(0L, 1L),
  n = 10L,
  s = c(0L, 0L),
  m = c(1L, 1L),
  inop = "+",
  form = 1L,
  rev = FALSE
)
```

Arguments

`inits`	a numeric (double or integer) vector of length `2`, giving the initial values. Any numbers are allowed, even negative and/or fractional numbers. Note that numbers given must give valid results when passed to function `f()`.
`n`	a single integer, giving the size of the numeric vector to generate. NOTE: it must hold that `n > 2`.
`s, m`	numeric vectors of length 2 to be used in the formula.
`inop`	a single string, giving the arithmetic infix operator to be used. Currently supported: "+", "-", "*", "/". For a fibonacci sequence, `inop = "+"`.
`form`	either `1` or `2`, indicating which formula to be used (see Description section above).
`rev`	reverse the order of `x[i-1]` and `x[-2]`. For example, using `form = 1`: • If `rev = FALSE` (default), it holds: `x[i] = (s[1] + m[1] * x[i-1]) %inop% (s[2] + m[2] * x[i-2])`. • If `rev = TRUE`, it holds: `x[i] = (s[1] + m[1] * x[i-2]) %inop% (s[2] + m[2] * x[i-1])`

Details

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

Value

A sequence of numbers.

Note

Do not supply `NA`s or `NaN`s to this function, as it cannot handle them.

Examples

```
seq_rec2() # by default gives Fibonacci numbers
seq_rec2(inits = 2:1) # Lucas numbers
c(1, seq_rec2(c(1, 2), inop = "*")) # Multiplicative Fibonacci
seq_rec2(m = c(2L, 1L)) # Pell numbers
seq_rec2(inits = c(1, 0), m = c(0L, 2L)) # see https://oeis.org/A077957
seq_rec2(m = c(1L, 2L)) # Jacobsthal numbers
```

---

setapply — *Apply Functions Over mutable_atomic Matrix Margins By Reference*

Description

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

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

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

Usage

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

Arguments

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

Value

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

Examples

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


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

`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 6 dimensions).

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

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

Usage

```
sub2coord(sub, x.dim)

coord2sub(coord)
```

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

ind2coord(ind, x.dim)

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

Arguments

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

Details

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

Arguments

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

Value

An integer vector of casted indices.

Examples

```
x.dim <- c(10, 10, 3)
x.len <- prod(x.dim)
x <- array(1:x.len, x.dim)
sub <- list(c(4, 3), c(3, 2), c(2, 3))
coord <- sub2coord(sub, x.dim)
```

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

Index