Package 'subsets'

December 30, 2023

Type Package

Title A Holistic Grammar of Subsets

Version 0.0.0.9

Date 2023-11-22

Description Subsetting methods for various data-types.
These methods have the following properties.
1) Programmatically friendly.
2) Beginner friendly.
3) Class consistent.
4) Careful handling of names and other attributes.
5) Performance aware.
6) Support atomic objects (vectors, matrices, arrays).
7) Support factors.
8) Support lists.
9) Support data.frame-like objects (data.frame, data.table, tibble, tidytable).
10) Support ggplot2 aesthetics.

License MIT + file LICENSE

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.3

Suggests rlang,
knitr,
rmarkdown,
tinytest,
abind,
tidytable,
tibble,
ggplot2,
sf,
tinycodet

Depends R (>= 4.2.0)

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

Encoding UTF-8

Language en-GB

VignetteBuilder knitr

R topics documented:

`aaa0_subsets` *subsets: an Easy Grammar of Subsets*

Description

subsets: an Easy Grammar of Subsets

Motivation

```
Among programming languages, 'R' has perhaps one of the most
flexible and comprehensive sub-setting functionality. But with
flexibility often comes confusion and (apparent) inconsistencies.
And 'R' is no exception.

This becomes quite apparent when one reads (online) documents such
as "The R Inferno" by Patrick Burns, and "Frustration: One Year
With R" by Reece Goding. These documents point out many
inconsistencies, and sub-setting related inconsistencies make up a
good portion of these documents.
```

```
To my surprise, there is no comprehensive R-package
(as far as I could see at least) that actually attempts to "fix" the
subset-related issues laid out in these and other such documents.

Famous subset-related R packages such as 'dplyr' and 'data.table'
focus almost exclusive on data.frame-like objects, and
occasionally even add more frustration in some aspects, like being
not very programmatically friendly.

Thus, this R package was born.

Although this package was somewhat made for people who are new to
'R' (especially when coming from another programming language),
and found themselves confused, I trust this package will be useful
even for those who are quite experienced in 'R'.
```

Goal & Properties

The Goal of the 'subsets' package is not to replace the square-brackets operators per-sé, (see Extract), but to provide **alternative** sub-setting methods and functions, to be used in situations where the square-brackets operators are inconvenient.
These are (hopefully) easier sub-setting methods and functions with the following properties:

- **Programmatically friendly**:
  - Non-standard evaluation is quite controversial (and for good reasons), and therefore completely absent in this R package.
  - Name-based arguments instead of position-based arguments.
  - Missing arguments can be filled with `NULL`, instead of using dark magic like `base::quote(expr = )`.
  - Functions are pipe-friendly.
- **Beginner friendly**:
  - No (silent) vector recycling.
  - Extracting and removing subsets uses the same syntax.
  - All functions return a copy of the object, unless stated otherwise.
- **Class consistent**:
  - sub-setting of multi-dimensional objects by specifying dimensions (i.e. rows, columns, ...) use `drop = FALSE`. So matrix in, matrix out.
  - The functions deliver the same results for data.frames, data.tables, tibbles, and tidytables. No longer does one have to re-learn the different brackets-based sub-setting rules for different types of data.frame-like objects. Powered by the subclass agnostic 'C'-code from 'collapse' and 'data.table'.
- **Explicit copy semantics**:
  - Sub-set operations that change its memory allocations, always return a modified copy of the object.
  - For sub-set operations that just change values in-place (similar to the `[<-` and `[[<-` methods) the user can choose a method that modifies the object by **reference**, or choose a method that returns a **deep copy**.
- **Careful handling of names and other attributes**:

  - Sub-setting an object by index names returns ALL indices with that name, not just the first.
  - Data.frame-like objects (see supported classes below) are forced to have unique column names.
  - Attributes of data.frame-like objects (see supported classes below) are always preserved when sub-setting.
  - For other object types, the user can specify whether to preserve Attributes, or use R's `[` attribute behaviour (i.e. drop most attributes). This is to ensure compatibility with R-packages that create their own attribute behaviour for sub-setting.
- **Support a wide variety of S3 classes**:
  - Support atomic objects (vectors, matrices, arrays).
  - Support factors.
  - Support lists.
  - Support the following data.frame-like objects: data.frame, data.table, tibble, and tidytable class, and classes that are straight-forward inheritors from these classes (such as sf-data.frames or sf-data.tables).
  - Support for the column selection sub-setting used in ggplot2's aes function.
  - Support for sub-setting characters in a single string.
  - Since the main functions are S3 methods, other packages may add functionality for additional S3 classes.
- **Concise function and argument names**.
- **Performance aware**:
  Despite the many checks performed, the functions are kept reasonably speedy, through the use of the 'Rcpp', 'collapse', and 'data.table' R-packages. Most of the heavy lifting in this package is done by the 'collapse' package.

Methods and Functions

The main focus is on the following generic S3 methods:

- sb_x: S3 method to extract, exchange, or duplicate subsets.
- sb_rm: S3 method to un-select ("remove") subsets.
- sb_set: S3 method to modify (transform or replace values) subsets of an object by **reference**. Since it's by reference, it does not allow coercion.
- sb_mod: S3 method to return a **copy** of an object with modified (transformed or replaced values) subsets. Has auto-coercion to a certain extent.
- sb_coe: S3 method to coerce and transform a whole object, or a recursive subset of an object.
- sb_before, sb_after: S3 methods to insert new values before or after an index along a dimension of an object.
- sb_rec: a function to accesses recursive subsets. Can be combined with the above S3 methods, for recursive sub-setting operations.

Additional specialized sub-setting functions are provided:

- aes_pro and with_pro: programmatically friendly and stable version of the with and `ggplot2::`aes functions.

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

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

- n: Nested version of c, and short-hand for list.
- idx_by: Compute grouped indices.
- The idx_ord_-functions: Compute ordered indices.
- seq_rec: Recursive sequence generator (for example to generate a Fibonacci sequence).
- seq_names: create a range of indices from a specified starting and ending name.
- sub2coord, coord2ind: Convert subscripts (array indices) to coordinates, coordinates to flat indices, and vice-versa.

Help pages

For an explanation of the classes, and how each class is treated by 'subsets', see subsets_classes.

For an explanation of the common indexing arguments in the generic methods, see subsets_indx_args.

Author(s)

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

References

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

See Also

'subsets' relies on the 'Rcpp', 'collapse' and 'data.table' R-packages to ensure an acceptable performance of its functions despite the many checks that these functions perform. I also recommend using these packages for other sub-setting and data wrangling functionalities.

'subsets' uses a modified version of the abind function from the 'abind' R-package; the 'abind' package is recommended for binding and sub-filling arrays of arbitrary dimensions.

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

- 'stringi':
  The primary R package for fast and concise string manipulation - an essential part of any programming language.

- 'tinycodet':
  Helps the user with their coding etiquette. Focuses on 4 aspects: (1) safe functionalities; (2) an import system that combines benefits of using a package without attaching, and attaching a package; (3) extending the capabilities of the 'stringi' package; (4) functions to reduce repetitive code.

`aaa1_subsets_classes` *Supported S3 classes, With Important Comments*

Description

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

- atomic vector classes (vectors, matrices, arrays);
- factors;
- lists;
- the data.frame-like classes data.frames, data.tables, tibbles, tidytables,
  and classes that are straight-forward inheritors from these classes (such as sf-data.frames or sf-data.tables).

These categories of classes are quite different from each other. It is vital to understand their differences, in order to use the 'subsets' package properly.
Thus this help page will explain their important properties and important differences.

Auto-Coercion Rules

Coercion Semantics

The sb_mod method modify subsets of an object through a **deep copy**.
The sb_set method and dt_setcoe function modify subsets of an object **by reference**.
These 2 copy semantics - "pass by reference" or "modify copy" - have slightly different auto-coercion rules. These are explained in this section.
Note that the sb_before and sb_after methods usually allow coercion for all classes.

Atomic

coercion_through_copy: YES
coercion_by_reference: NO
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`.

Replacing or transforming subsets of 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.

Factor

coercion_through_copy: NO

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

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

List

coercion_through_copy: depends

coercion_by_reference: depends

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

This is true regardless if replacement/transformation takes place by reference or through a copy.

For example, the following code performs full coercion:

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

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) |> sb_mod(i = 1, rp = "a") # coerces to character
sb_rec(x, 1) |> sb_set(i = 1, rp = "a"); print(x) # no coercion; "a" replaced with NA.
```

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

Data.frame-like, when replacing/transforming whole columns

coercion_through_copy: YES

coercion_by_reference: YES

A data.frame-like objects (data.frames, data.tables, tibbles, tidytables, and their sf-equivalents) are actually lists, 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.frame-like, 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()).

Technical Details

Atomic

require unique names: NO
The atomic vector is the most basic class type.
Matrices and Arrays are just atomic vectors with dimension attributes.
All elements in an atomic vector class must have the same atomic type ("logical", "integer", "numeric", "complex", "character" and "raw").
Therefore, the sb_coe method will coerce the entirety of an atomic vector, NOT just a subset of it.

Factor

require unique names: NO
Factors have the property that they can only store values that are defined in their "levels" attribute; other values are not allowed, and thus result in NAs.
Thus And sb_mod does NOT coerce replacement values for factors! This is quite different from the atomic vector classes, which can store an infinite variety of values (provided they are of the same atomic type).
Due to these properties, there is NO sb_set() method for factors.

List

require unique names: NO
Lists are recursive objects (i.e. they can be nested), and they do not actually store values but rather store reference to other objects.
Therefore sb_rec method can be used to access recursive subsets of a list, no matter how deep/low in hierarchy it is in the list.

Data.frame-like

require unique names: YES
The data.frame-like objects quite different from the previously named classes.
And the different data.frame-like classes also differ from each other quite a bit - especially in terms of sub-setting.
The 'subsets' R-package attempts to keep the data.frame methods as class agnostic as possible, through the class agnostic functionality of the 'collapse' and 'data.table' R-packages.
These 3 things cause some **important** oddities in how data.frame-like classes are treated differently from the other classes:

- Whole-columns will be auto-coerced when replaced/transformed by sb_mod(), but partial columns will NOT be auto-coerced.
- The sb_x and sb_rm methods always automatically conserve all attributes (though names are adjusted accordingly, of course), they are never stripped, unlike the other classes.
- Giving a data.frame-like object with non-unique column names to the sb_-methods returns an error. Also, duplicating columns with sb_x will automatically adjust the column names to make them unique.

Examples

```
# Coercion examples - Atomic ====

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

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


# Coercion examples - List ====

x <- list(factor(letters), factor(letters))
sb_mod(x, 1, rp = list(1))
sb_set(x, 1, rp = list(1)); print(x)
x <- list(1:10, 1:10)

sb_rec(x, 1) |> sb_mod(i = 1, rp = "a") # coerces to character
sb_rec(x, 1) |> sb_set(i = 1, rp = "a"); print(x) # no coercion; "a" replaced with NA.

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


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

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

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

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


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

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

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

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

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

`aaa2_subsets_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 'subsets' to specify the indices to perform operations on:

- `i`: to specify flat (i.e. dimensionless) indices.
- `row, col`: to specify rows and/or columns in tabular objects.
- `idx, dims`: to specify indices of arbitrary dimensions in arrays.
- `rcl`: to specify rows (first dimension), columns (second dimension), and layers (third dimension), in arrays that have exactly 3 dimensions.
- `lvl`: specify levels, for factors only.
- `filter, vars`: to specify rows and/or columns specifically in data.frame-like objects.

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

Argument i

class: atomic
class: factor
class: list

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

- `NULL`, only for multi-dimensional objects or factors, when specifying the other arguments (i.e. dimensional indices or factor levels.)
- a vector of length 0, in which case no indices are selected for the operation (i.e. empty selection).
- a **strictly positive integer** vector with indices.
- a **logical vector** (without `NA`s!), of the same length as `x`, giving the indices to select for the operation.
- a **character** vector of index names.
  If an object has multiple indices with the given name, ALL the corresponding indices will be selected for the operation.
- a **function** that takes as input `x`, and returns a logical vector, giving the element indices to select for the operation.
  For atomic objects, `i` is interpreted as `i(x)`.
  For lists, `i` is interpreted as `lapply(x, i)`.

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

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

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

Arguments row, col

class: matrix
class: data.frame-like

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

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

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

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

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

Arguments idx, dims

class: array

`idx` must be a list of indices.
`dims` must be an integer vector of the same length as `idx`, giving the dimensions to which the indices given in `idx` correspond to.
The elements of `idx` follow the same rules as the rules for `row` and `col`, EXCEPT one should not fill in `NULL`.
NOTE: The arguments `idx` and `dims` will be ignored if `i` is specified.

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

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

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

Arguments rcl

class: array

The `rcl` argument is only applicable for arrays with exactly 3 dimensions.
If the user knows a-priori that an array has 3 dimensions, using `rcl` is more efficient than using the `idx, dims` arguments.

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

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

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

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

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

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

Argument lvl

class: factor

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

Arguments filter, vars

class: data.frame-like

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

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

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

Duplicates (for Names, Integers, and Levels)

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

Out-of-Bounds Integers and Unknown Names/Levels

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

Disallowed Combinations of Index Arguments

One cannot specify `i` and the other indexing arguments simultaneously; it's either `i`, or the other arguments.
The arguments are evaluated in the following order:

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

One cannot specify `row` and `filter` simultaneously. It's either one or the other. Similarly, one cannot specify `col` and `vars` simultaneously.
In the above cases it holds that if one set is specified, the other is set is ignored.

Drop

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

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

First, Last, and Shuffle

The indices are counted forward. I.e. `1` is the first element, not the last.
One can use the last function to get the last `N` indices.

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

To shuffle elements of indices, use the sample function.

`aaa3_subsets_misc` *Miscellaneous Information*

Description

Miscellaneous Information

Controversy Surrounding Non-Standard Evaluation

Non-Standard Evaluation (sometimes abbreviated as "NSE"), is quite controversial.
Consider the following example:

```
aplot <- "ggplot2"
library(aplot)
```

What package will be attached? It will not be 'ggplot2', nor will an error occur. Instead, the package 'aplot' will be attached.
This is due to evaluating the expression 'aplot' as a quoted expression, instead of evaluating the contents (i.e. string or formula) of the variable. In other words: Non-Standard Evaluation.

A standard evaluated expression works better.
Standard evaluation in 'R' is not limited to characters. Formulas can also be used. For example, if the `library()` function would support a formula input, the following would correctly load 'ggplot2':

```
aplot <- ~ ggplot2
library(aplot)
```

Performance Tips

A few tips to improve performance:

- Use integer vectors for sub-setting whenever possible.
- Do not set `rat = TRUE` if it's not truly necessary (it defaults to `rat = FALSE`).
- Perform as much sub-set operations in a single function.

`aes_pro` *Programmatically Friendly, Standard Evaluated Versions of with() and ggplot2::aes()*

Description

Programmatically friendly versions of the with and `ggplot2::aes` functions. These alternative functions are more programmatically friendly because it uses proper standard evaluation, instead of non-standard evaluation, tidy evaluation, or similar programmatically unfriendly evaluations.

Usage

```
aes_pro(...)

with_pro(data, form)
```

Arguments

`...`	arguments to be passed to aes, but given as one-sided formulas, rather than non-standard evaluated dark magic.
`data`	a list, environment, or data.frame.
`form`	a one-sided formula giving the expression to evaluate. Global variables are not allowed, only variables that are actually present in the data.

Details

Non-Standard Evaluation is quite controversial (see subsets_misc).

Often standard-evaluated alternatives provided.
But in the case of the `aes()` function in 'ggplot2', the standard-evaluated alternative changes frequently, and the ones provided so far are rather clumsy.

The `aes_pro()` function is the standard evaluated alternative. Due to the way `aes_pro()` is programmed, it should work no matter how many times the standard evaluation techniques change in 'ggplot2'.
It should also work in older and newer versions of 'ggplot2'.
To support functions in combinations with references of the variables, the input used here are formula inputs, rather than character inputs.
See Examples section below.

Value

See aes.

Examples

```
requireNamespace("ggplot2")


data("mpg", package = "ggplot2")
x <- ~ cty
y <- ~ sqrt(hwy)
color <- ~ drv

ggplot2::ggplot(mpg, aes_pro(x, y, color = color)) +
  ggplot2::geom_point()

myform <- ~ sqrt(hwy)
mpg$hwy_sqrt <- with_pro(mpg, myform)
summary(mpg)
```

---

dt *Functional forms of data.table Operations (also work on tidytables)*

Description

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

`dt_aggregate()` aggregates a data.table or tidytable, and returns the aggregated copy.
`dt_setcoe()` coercively transforms columns of a data.table or tidytable BY REFERENCE.

Usage

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

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

dt_setrm(x, col = NULL, vars = 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	logical (TRUE or FALSE), indicating if the aggregated result should be ordered by the columns specified in by.
col, vars	columns to select for coercion; see subsets_indx_args. Duplicates are not allowed.

Value

The sub-setted object.

Examples

```
requireNamespace("sf") && requireNamespace("ggplot2")


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

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

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

ggplot2::ggplot(plotdat, aes_pro(geometry = ~ geometry, fill = ~ region)) +
  ggplot2::geom_sf()

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


# dt_setcoe ====

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

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



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


# dt_setrm ====

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

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

`idx_by` *Compute Grouped Indices*

Description

Given:

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

the `idx_by()` function takes indices `f(r)` **per group** `grp`.
The result of `idx_by()` can be supplied to the indexing arguments (see subsets_indx_args) of:
sb_x, sb_rm, sb_mod, sb_set, or sb_coe,
to perform **grouped** subset operations.

Usage

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

Arguments

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

Value

A vector of indices of the same type as r.

Examples

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

# get the last element of `a` for each group in `grp`:
i <- idx_by(last, 1:length(a), grp)
sb_x(cbind(a, grp), row = i)


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

---

idx_ord_v *Compute Ordered Indices*

Description

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

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

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

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

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

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

Usage

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

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

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

Arguments

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

Value

See order.

Examples

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

`indx_x` *Exported Utilities*

Description

Exported utilities

Usage

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

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

Arguments

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

Value

The subsetted object.

Examples

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

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 'subsets', 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]
```

safer_partialmatch *Set safer dollar, arguments, and attribute matching*

Description

This function simply calls the following:

```
options(
   warnPartialMatchDollar = TRUE,
   warnPartialMatchArgs = TRUE,
   warnPartialMatchAttr = TRUE
 )
```

Thus it forces 'R' to give a warning when partial matching occurs when using the dollar ($) operator, or when other forms of partial matching occurs.

Usage

```
safer_partialmatch()
```

Value

Sets the options. Returns nothing.

Examples

```
interactive()


safer_partialmatch()
data(iris)
head(iris)
iris$Sepal.Length <- iris$Sepal.Length^2
head(iris)
```

---

sb_before *Methods to Insert New Values Before or After an Index Along a Dimension*

Description

The sb_before() method inserts new values before some position along a dimension.
The sb_after() method inserts new values after some position along a dimension.

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

Usage

```
sb_before(x, ...)

sb_after(x, ...)

## Default S3 method:
sb_before(x, new, pos = 1, .attr = NULL, ...)

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

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

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

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

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

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

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

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

## S3 method for class 'data.frame'
sb_after(x, new, margin, pos = collapse::fdim(x)[margin], .attr = NULL, ...)
```

Arguments

x	see subsets_classes.
...	further arguments passed to or from other methods.
new	the new value(s). The type of object depends on x: • For atomic objects, new can be any atomic object. However, if one wished the added values in new to be named, ensure new is the same type of object as x. For example: use matrix with column names for new when appending/inserting columns to matrix x. • For factors, new must be a factor. • For lists, new must be a (possible named) list. • For data.frame-like objects, new must be a data.frame.
pos	a strictly positive single integer scalar (so no duplicates), giving the position along the dimension (specified in margin), before or after which the new values are added.
.attr	a list, giving additional potentially missing attributes to be added to the returned object. By default, concatenation strips attributes, since the attributes of x and new may not be equal or even compatible. In the attr argument, the attributes of the merged object can be specified. Only attributes that are actually missing AFTER insertion will be added, thus preventing overwriting existing attributes like names. One may, for example, specify .attr = sb_a(x) or .attr = sb_a(new). If NULL (the default), no attributes will be added post-insert. If speed is important, NULL is the best option (but then attributes won't be preserved).
margin	a single scalar, giving the dimension along which to add new values.

Value

Returns a copy of the appended object.

References

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

Examples

```
# atomic objects ====

x <- matrix(1:20 , ncol = 4)
print(x)
new <- -1 * x
sb_before(x, new, 1)
sb_before(x, new, 2)
sb_after(x, new, 1)
sb_after(x, new, 2)

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


# factors ====

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

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


# lists ====

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

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


# data.frame-like objects ====

x <- data.frame(a = 1:10, b = letters[1:10], c = 11:20, d = factor(letters[1:10]))
new <- data.frame(e = 101:110)
sb_before(x, new, 2)
sb_after(x, new, 2)
new <- x[1,]
sb_before(x, new, 1)
sb_after(x, new, 1)
```

---

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

Description

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

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

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

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

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

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

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

```
x[i] <- lapply(x[i], v)
```

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

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

Note that when `x` is a `data.table`, one can coercively transform columns BY REFERENCE (which is more memory efficient), using the following code (again with columns specified by `col`, and some coercive transformation function `v`):

```
col <- ... # some integer/character vector of column indices/names
x[, (col) := lapply(.SD, v), .SDcols = col]
```

Usage

```
sb_coe(x, ...)

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

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

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

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

Arguments

`x`	see subsets_classes.
`...`	further arguments passed to or from other methods.
`v`	the coercive transformation function to use.
`i, col, vars`	See subsets_indx_args. An empty index selection returns the original object unchanged.

Details

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

For example:
Using `sb_set()` to replacing/transform one or more values of an integer type (`int`) object / list element / data.frame column, to become `1.5`, will NOT coerce the object / list element / data.frame column to a decimal type (`dbl`); instead, the replacement `1.5` is coerced to the integer `1`.

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

Value

A copy of the coercively transformed object.

Examples

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

---

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

Description

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

Usage

```
sb_mod(x, ...)

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

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

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

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

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

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

Arguments

`x`	see subsets_classes.
`...`	further arguments passed to or from other methods.
`i, lvl, row, col, idx, dims, rcl, filter, vars`	See subsets_indx_args. An empty index selection returns the original object unchanged.
`rp`	an object of somewhat the same type as the selected subset of `x`, and the same same length as the selected subset of `x` or a length of 1.
`tf`	the transformation function.
`coe`	For data.frame-like objects, `sb_mod()` can only coerce whole columns, not subsets of columns. So it does not automatically coerce column types when `row` or `filter` is also specified. Therefore, the user can specify a coercion function, to be applied on the entirety of every column specified in `col` or `vars`; columns outside this subset are not affected. This coercion function is, of course, applied before replacement (`rp`) or transformation (`tf()`). By default, `coe = NULL` which means no columns are coercively transformed. See also the `Auto-Coercion Rules` and `Examples` sections below, and also see sb_coe.

Details

Transform or Replace

Specifying argument `tf` will transform the subset.

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

Value

A copy of the object with replaced/transformed values.

Examples

```
# atomic objects ====

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

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

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

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


# lists ====

obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_mod(obj, "a", rp = list(1L))
```

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

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


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

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

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

`sb_rec` *Access Recursive Subsets*

Description

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

Usage

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

Arguments

`lst`	a list, or list-like object.
`rec`	a vector of length p, such that `lst[[rec]]` is equivalent to `lst[[ rec[1] ]]...[[ rec[p] ]]`, providing all but the final indexing results in a list. When on a certain subset level of a nested list, multiple subsets with the same name exist, only the first one will be selected when performing recursive indexing by name, due to the recursive nature of this type of subsetting.

Value

The sub-setted object.

Examples

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

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

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

---

sb_rm *Method to Un-Select Subsets of an Object*

Description

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

Usage

```
sb_rm(x, ...)

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

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

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

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

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

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

Arguments

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

Details

One the `rat` argument

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

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

Value

A copy of the sub-setted object.

Examples

```
# atomic objects ====

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_rm(obj, 1:3, 1:3)
```

```
# above is equivalent to  obj[-1:-3, -1:-3, drop = FALSE]
sb_rm(obj, i = \(x)x>5)
# above is equivalent to  obj[!obj > 5]
sb_rm(obj, col = "a")
# above is equivalent to  obj[, which(!colnames(obj) %in% "a")]

obj <- array(1:64, c(4,4,3))
print(obj)
sb_rm(obj, n(1, c(1, 3)), c(1, 3))
sb_rm(obj, rcl = n(1, NULL, c(1, 3)))
# above 2 lines are equivalent to obj[-1, c(-1, -3), drop = FALSE]
sb_rm(obj, i = \(x)x>5)
# above is equivalent to obj[!obj > 5]

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


# lists ====

obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_rm(obj, "a")
# above is equivalent to obj[which(!names(obj) %in% "a")]
sb_rm(obj, 1) # obj[-1]
sb_rm(obj, 1:2)
# above is equivalent to obj[seq_len(length(obj))[-1:-2]]
sb_rm(obj, is.numeric, drop = TRUE)
# above is equivalent to obj[[!sapply(obj, is.numeric)]] IF this returns a single element
obj <- list(a = 1:10, b = letters[1:11], c = letters)
sb_rm(obj, is.numeric)
# above is equivalent to obj[!sapply(obj, is.numeric)] # this time singular brackets?
# for recusive indexing, see sb_rec()

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


# factors ====

obj <- factor(rep(letters[1:5], 2))
sb_rm(obj, lvl = "a")
# above is equivalent to obj[which(!obj %in% "a")]

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


# data.frame-like objects ====

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

sb_set *Method to Modify Subsets of an Object By Reference*

Description

This is an S3 Method to replace or transform a subset of an object **By Reference**.

Usage

```
sb_set(x, ...)

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

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

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

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

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

Arguments

`x`	see subsets_classes.
`...`	further arguments passed to or from other methods.
`i, row, col, idx, dims, rcl, filter, vars`	See subsets_indx_args. An empty index selection returns the original object unchanged.
`rp`	an object of somewhat the same type as the selected subset of `x`, and the same same length as the selected subset of `x` or a length of 1.
`tf`	the transformation function.

Details

Transform or Replace

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

Note that there is not `sb_set()` method for factors: this is intentional.

Value

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

Reference Semantics

The `sb_set()` method modifies an object **by reference**.
The advantage of this is that less memory is required to modify objects.
But modifying an object by reference does have 3 potential disadvantages.

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

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

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

modifies not just `x`, but also `y`.

Third, the second consequence is true even if one of the variables is locked (see bindingIsLocked).
I.e. the following code,

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

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

Examples

```
# atomic objects ====

gen_mat <- function() {
  obj <- 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() {
  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.frame ====

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

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

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

---

sb_special *Specialized Sub-setting Functions*

Description

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

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

Usage

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

sb_a(x, a = NULL)
```

Arguments

str	a single string.
ind	an integer vector, giving the positions of the string to subset.
rp.str, rp.ind	similar to str and ind, respectively. If not specified, 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. replicate indices x times) indices of an object.

Usage

```
sb_x(x, ...)

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

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

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

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

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

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

Arguments

x	see subsets_classes.
...	further arguments passed to or from other methods.

`i, lvl, row, col, idx, dims, rcl, filter, vars`
: See subsets_indx_args.
Duplicates are allowed, resulting in duplicated indices.
An empty index selection results in an empty object of length 0.

`rat`
: logical, indicating if attributes should be returned with the sub-setted object. See Details section for more info.

`drop`
: logical.
- For factors: If `drop = TRUE`, unused levels are dropped, if `drop = FALSE` they are not dropped.
- For lists: if `drop = TRUE`, selecting a single element will give the simplified result, like using `[[]]`. If `drop = FALSE`, a list is always returned regardless of the number of elements.

Details

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

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

Value

Returns a copy of the sub-setted object.

Examples

```
# atomic objects ====

obj <- matrix(1:16, ncol = 4)
colnames(obj) <- c("a", "b", "c", "a")
print(obj)
sb_x(obj, 1:3, 1:3)
# above is equivalent to obj[1:3, 1:3, drop = FALSE]
sb_x(obj, i = \(x)x>5)
# above is equivalent to obj[obj > 5]
sb_x(obj, col = c("a", "a"))
# above is equivalent to obj[, lapply(c("a", "a"), \(i) which(colnames(obj) == i)) |> unlist()]

obj <- array(1:64, c(4,4,3))
print(obj)
sb_x(obj, n(1:3, 1:2), c(1,3))
sb_x(obj, rcl = n(1:3, NULL, 1:2))
# above 2 lines are equivalent to obj[1:3, , 1:2, drop = FALSE]
sb_x(obj, i = \(x)x>5)
# above is equivalent to obj[obj > 5]
```

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


# lists ====

obj <- list(a = 1:10, b = letters[1:11], c = 11:20)
print(obj)
sb_x(obj, 1) # obj[1]
sb_x(obj, 1, drop = TRUE) # obj[[1]]
sb_x(obj, 1:2) # obj[1:2]
sb_x(obj, is.numeric) # obj[sapply(obj, is.numeric)]
# for recursive indexing, see sb_rec()

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


# factors ====

obj <- factor(rep(letters[1:5], 2))
sb_x(obj, lvl = c("a", "a"))
# above is equivalent to obj[lapply(c("a", "a"), \(i) which(obj == i)) |> unlist()]

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


# data.frame-like objects ====

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

`seq_names` *Generate Integer Sequence From a Range of Names*

Description

Generate integer sequence from a range of names.

Usage

```
seq_names(names, start, end, inv = FALSE)
```

Arguments

`names`	a character vector of names. Duplicate names, empty names, or a character vector of length zero are not allowed.
`start`	the name giving the starting index of the sequence
`end`	the name giving the ending index of the sequence
`inv`	logical, if TRUE, the indices of all names EXCEPT the names of the specified sequence will be given.

Value

An integer vector.

Examples

```
x <- data.frame(a = 1:10, b = letters[1:10], c = factor(letters[1:10]), d = -1:-10)
ind <- seq_names(colnames(x), "b", "d")
sb_x(x, col = ind)
```

---

seq_rec *Recursive Sequence Generator*

Description

This is a recursive sequence generator. The function is essentially a highly generalized version of a Fibonacci sequence generator. One can change the initial values, the window size, and even the window function used.
This function assumes only the following about the sequence being generated:

- The sequence consists of real numbers (i.e. class `integer` or class `double`).
- The window size is the same for all iterations.
- The window function is the same for all iterations.
- The sequence grows until a vector of length `n` is achieved.

Usage

```
seq_rec(inits = c(0, 1), n = 10L, f = sum)
```

Arguments

`inits`	a numeric (double or integer) vector giving the initial values. Any numbers are allowed, even negative and/or fractional numbers. Note that numbers given must give valid results when passed to function `f()`. IMPORTANT: The length of `inits` determines the window size `w`. For a regular Fibonacci, `inits = 0:1`, which of course means a window size of `w = 2`.
`n`	a single integer, giving the size of the numeric vector to generate. NOTE: it must hold that `n` is larger than or equal to the window size `w`. The window size is equal to `w = length(inits)`.
`f`	the function to be used on the last `w` numbers to generate the next number of the sequence at each iteration. This must be a function that takes as input a numeric vector, and returns a single numeric value. For a regular Fibonacci sequence, this would be either: `f = sum`, or (since window size is 2) `f = \(x) x[2] + x[1]`

Details

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

Value

A sequence of numbers.

Examples

```
seq_rec() # by default gives Fibonacci numbers
seq_rec(0:3, 10L, sum) # a weird shifted version of Fibonacci
seq_rec(inits=2:1) # Lucas numbers
c(1, seq_rec(c(1, 2), f=prod)) # Multiplicative Fibonacci
seq_rec(f=\(x)2*x[2]+x[1]) # Pell numbers
seq_rec(inits = c(1, 0), f=\(x)2*x[1]) # see https://oeis.org/A077957
seq_rec(f=\(x)x[2]+2*x[1]) # Jacobsthal numbers
seq_rec(c(1,1,1), f=\(x)x[1] + x[2]) # Padovan sequence
seq_rec(c(3,0,2), f=\(x)x[1] + x[2]) # Perrin numbers
seq_rec(c(0,1,3), f=\(x)3*x[3] - 3*x[2] + x[1]) # Triangular numbers
```

---

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

Description

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

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

Usage

```
sub2coord(sub, x.dim)

coord2sub(coord)

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

ind2coord(ind, x.dim)
```

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	logical, indicating if arguments checks should be performed. Defaults to TRUE. Can be set to FALSE for minor speed improvements, but not recommended.
ind	an integer vector, giving the flat position indices to convert.

Details

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

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

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

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

The subscript [1,2] refers to the first row and the second column. In a 4 by 4 matrix, subscript [1,2] corresponds to flat index 5.
The functions described here thus follow also this convention.

Value

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

For coord2ind():
Returns an integer vector of flat indices (with properties as described in argument ind).

For `coord2sub()`:
Returns a list of integer subscripts (with properties as described in argument `sub`)

Examples

```
x.dim <- c(1000, 10, 4, 4)
x.len <- prod(x.dim)
x <- array(1:x.len, x.dim)
sub <- list(c(4,1), c(3,2), c(2,3), c(1,4))
coord <- sub2coord(sub, x.dim)
print(coord)
ind <- coord2ind(coord, x.dim)
print(ind)
all(x[ind] == c(x[c(4,1), c(3,2), c(2,3), c(1,4)])) # TRUE
coord2 <- ind2coord(ind, x.dim)
print(coord)
all(coord == coord2) # TRUE
sub2 <- coord2sub(coord2)
sapply(1:4, \(i)sub2[[i]]==sub[[i]]) |> all() # TRUE
```

Index