Package ‘cheapr’

October 2, 2024

Title Simple Functions to Save Time and Memory

Version 0.9.8

Maintainer Nick Christofides <nick.christofides.r@gmail.com>

Description Fast and memory-efficient (or 'cheap') tools to facilitate efficient programming, saving time and memory. It aims to provide 'cheaper' alternatives to common base R functions, as well as some additional functions.

License MIT + file LICENSE

BugReports https://github.com/NicChr/cheapr/issues

Depends R (>= 3.5.0)

Imports collapse (>= 2.0.0)

Suggests bench, data.table, testthat (>= 3.0.0)

LinkingTo cpp11

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation yes

Author Nick Christofides [aut, cre] (<https://orcid.org/0000-0002-9743-7342>)

Repository CRAN

Date/Publication 2024-10-02 10:50:04 UTC

Contents

`cheapr-package` *cheapr: Simple Functions to Save Time and Memory*

Description

In this package, 'cheap' means fast and efficient.

cheapr aims to provide a set of functions for programmers to write cheaper code, saving time and memory.

Author(s)

Maintainer: Nick Christofides <`nick.christofides.r@gmail.com`> (ORCID)

See Also

Useful links:

- Report bugs at `https://github.com/NicChr/cheapr/issues`

`as_discrete` *Turn continuous data into discrete bins*

Description

Turn continuous data into discrete bins

Usage

```
as_discrete(x, ...)

## S3 method for class 'numeric'
as_discrete(
  x,
  breaks = get_breaks(x, expand_max = TRUE),
  left_closed = TRUE,
  include_endpoint = FALSE,
  include_oob = FALSE,
  ordered = FALSE,
  intv_start_fun = prettyNum,
  intv_end_fun = prettyNum,
  intv_closers = c("[", "]"),
  intv_openers = c("(", ")"),
  intv_sep = ",",
  ...
)

## S3 method for class 'integer64'
as_discrete(x, ...)
```

Arguments

`x`	A numeric vector.
`...`	Extra arguments passed onto methods.
`breaks`	Break-points.
`left_closed`	Left-closed intervals or right-closed intervals?
`include_endpoint`	Include endpoint? Default is `FALSE`.
`include_oob`	Include out-of-bounds values? Default is `FALSE`.
`ordered`	Should result be an ordered factor? Default is `FALSE`.
`intv_start_fun`	Function used to format interval start points.
`intv_end_fun`	Function used to format interval end points.
`intv_closers`	A length 2 character vector denoting the symbol to use for closing either left or right closed intervals.
`intv_openers`	A length 2 character vector denoting the symbol to use for opening either left or right closed intervals.
`intv_sep`	A length 1 character vector used to separate the start and end points.

Value

A factor of discrete bins (intervals of start/end pairs).

Examples

```
library(cheapr)

# `as_discrete()` is very similar to `cut()`
# but more flexible as it allows you to supply
# formatting functions and symbols for the discrete bins

# Here is an example of how to use the formatting functions to
# categorise age groups nicely

ages <- 1:100

age_group <- function(x, breaks){
  age_groups <- as_discrete(
    x,
    breaks = breaks,
    intv_sep = "--",
    intv_end_fun = function(x) x - 1,
    intv_openers = c("", ""),
    intv_closers = c("", ""),
    include_oob = TRUE,
    ordered = TRUE
  )

  # Below is just renaming the last age group

  lvls <- levels(age_groups)
  n_lvls <- length(lvls)
  max_ages <- paste0(max(breaks), "+")
  attr(age_groups, "levels") <- c(lvls[-n_lvls], max_ages)
  age_groups
}

age_group(ages, seq(0, 80, 20))
age_group(ages, seq(0, 25, 5))
age_group(ages, 5)
```

`bin` *A sometimes cheaper but argument richer alternative to* `.bincode()`

Description

When `x` is an integer vector, `bin()` is cheaper than `.bincode()` as no coercion to a double vector occurs. This alternative also has more arguments that allow you to return the start values of the binned vector, as well as including out-of-bounds intervals.

Usage

```
bin(
  x,
  breaks,
  left_closed = TRUE,
  include_endpoint = FALSE,
  include_oob = FALSE,
  codes = TRUE
)
```

Arguments

`x`	A numeric vector.
`breaks`	A numeric vector of breaks.
`left_closed`	Should intervals be left-closed (and right-open)? Default is `TRUE`. If `FALSE` they are left-open (and right-closed).
`include_endpoint`	Equivalent to `include.lowest` in `?.bincode`.
`include_oob`	Should out-of-bounds interval be included? Default is `FALSE`. This is the equivalent of adding `Inf` as the last value of the breaks, or `-Inf` as the first value of the breaks if `left_closed = FALSE`.
`codes`	Should an integer vector indicating which bin the values fall into be returned? Default is `TRUE`. If `FALSE` the start values of the respective bin intervals are returned, i.e the corresponding breaks.

Value

Either an integer vector of codes indicating which bin the values fall into, or the start of the intervals for which each value falls into.

`factor_` *A cheaper version of* `factor()` *along with cheaper utilities*

Description

A fast version of `factor()` using the collapse package.

There are some additional utilities, most of which begin with the prefix 'levels_', such as `as_factor()` which is an efficient way to coerce both vectors and factors, `levels_factor()` which returns the levels of a factor, as a factor, `levels_used()` which returns the used levels of a factor, `levels_unused()` which returns the unused levels of a factor, `levels_add_na()` which adds an explicit `NA` level, `levels_drop_na()` which drops the `NA` level, `levels_drop()` which drops unused factor levels, and finally `levels_reorder()` which reorders the levels of `x` based on `y` using the ordered median values of `y` for each level.

Usage

```
factor_(
  x = integer(),
  levels = NULL,
  order = TRUE,
  na_exclude = TRUE,
  ordered = is.ordered(x)
)

as_factor(x)

levels_factor(x)

levels_used(x)

levels_unused(x)

used_levels(x)

unused_levels(x)

levels_add_na(x, name = NA, where = c("last", "first"))

levels_drop_na(x)

levels_drop(x)

levels_reorder(x, order_by, decreasing = FALSE)
```

Arguments

`x`	A vector.
`levels`	Optional factor levels.
`order`	Should factor levels be sorted? Default is `TRUE`. It typically is faster to set this to `FALSE`, in which case the levels are sorted by order of first appearance.
`na_exclude`	Should `NA` values be excluded from the factor levels? Default is `TRUE`.
`ordered`	Should the result be an ordered factor?
`name`	Name of `NA` level.
`where`	Where should `NA` level be placed? Either first or last.
`order_by`	A vector to order the levels of `x` by using the medians of `order_by`.
`decreasing`	Should the reordered levels be in decreasing order? Default is `FALSE`.

Details

This operates similarly to `collapse::qF()`.
The main difference internally is that `collapse::funique()` is used and therefore s3 methods can

be written for it.
Furthermore, for date-times `factor_` differs in that it differentiates all instances in time whereas `factor` differentiates calendar times. Using a daylight savings example where the clocks go back:
`factor(as.POSIXct(1729984360, tz = "Europe/London") + 3600 *(1:5))` produces 4 levels whereas
`factor_(as.POSIXct(1729984360, tz = "Europe/London") + 3600 *(1:5))` produces 5 levels.

Value

A `factor` or `character` in the case of `levels_used` and `levels_unused`.

---

gcd — *Greatest common divisor and smallest common multiple*

Description

Fast greatest common divisor and smallest common multiple using the Euclidean algorithm.

`gcd()` returns the greatest common divisor.
`scm()` returns the smallest common multiple.
`gcd2()` is a vectorised binary version of `gcd`.
`scm2()` is a vectorised binary version of `scm`.

Usage

```
gcd(
  x,
  tol = sqrt(.Machine$double.eps),
  na_rm = TRUE,
  round = TRUE,
  break_early = TRUE
)

scm(x, tol = sqrt(.Machine$double.eps), na_rm = TRUE)

gcd2(x, y, tol = sqrt(.Machine$double.eps), na_rm = TRUE)

scm2(x, y, tol = sqrt(.Machine$double.eps), na_rm = TRUE)
```

Arguments

`x`	A numeric vector.
`tol`	Tolerance. This must be a single positive number strictly less than 1.
`na_rm`	If `TRUE` the default, `NA` values are ignored.
`round`	If `TRUE` the output is rounded as `round(gcd, digits)` where digits is `ceiling(abs(log10(tol))) + 1`. This can potentially reduce floating point errors on further calculations. The default is `TRUE`.

`break_early`	This is experimental and applies only to floating-point numbers. When `TRUE` the algorithm will end once `gcd > 0 && gcd < 2 * tol`. This can offer a tremendous speed improvement. If `FALSE` the algorithm finishes once it has gone through all elements of `x`. The default is `TRUE`. For integers, the algorithm always breaks early once `gcd > 0 && gcd <= 1`.
`y`	A numeric vector.

Details

Method:

GCD (Greatest Common Divisor):
The GCD is calculated using a binary function that takes input `GCD(gcd, x[i + 1])` where the output of this function is passed as input back into the same function iteratively along the length of `x`. The first gcd value is `x[1]`.
Zeroes are handled in the following way:
`GCD(0, 0) = 0`
`GCD(a, 0) = a`

This has the nice property that zeroes are essentially ignored.

SCM (Smallest Common Multiple):
This is calculated using the GCD and the formula is:
`SCM(x, y) = (abs(x) / GCD(x, y) ) * abs(y)`
If you want to calculate the gcd & lcm for 2 values or across 2 vectors of values, use `gcd2` and `scm2`.

A note on performance:
A very common solution to finding the GCD of a vector of values is to use `Reduce()` along with a binary function like `gcd2()`.
e.g. `Reduce(gcd2, seq(5, 20, 5))`.
This is exactly identical to `gcd(seq(5, 20, 5))`, with `gcd()` being much faster and overall cheaper as it is written in C++ and heavily optimised. Therefore it is recommended to always use `gcd()`.
For example we can compare the two approaches below,
`x <- seq(5L, length = 10^6, by = 5L)`
`bench::mark(Reduce(gcd2, x), gcd(x))`
This example code shows `gcd()` being ~200x faster on my machine than the `Reduce + gcd2` approach, even though `gcd2` itself is written in C++ and has little overhead.

Value

A number representing the GCD or SCM.

Examples

```
library(cheapr)
library(bench)

# Binary versions
gcd2(15, 25)
```

```
gcd2(15, seq(5, 25, 5))
scm2(15, seq(5, 25, 5))
scm2(15, 25)

# GCD across a vector
gcd(c(0, 5, 25))
mark(gcd(c(0, 5, 25)))

x <- rnorm(10^5)
gcd(x)
gcd(x, round = FALSE)
mark(gcd(x))
```

`get_breaks` *Pretty break-points for continuous (numeric) data*

Description

The distances between break-points are always equal in this implementation.

Usage

```
get_breaks(x, n = 7, ...)

## S3 method for class 'numeric'
get_breaks(
  x,
  n = 7,
  pretty = TRUE,
  expand_min = FALSE,
  expand_max = FALSE,
  ...
)

## S3 method for class 'integer64'
get_breaks(x, n = 7, ...)
```

Arguments

`x`	A numeric vector.
`n`	Number of breakpoints. You may get less or more than requested.
`...`	Extra arguments passed onto methods.
`pretty`	Should pretty break-points be prioritised? Default is `TRUE`.
`expand_min`	Should smallest break be extended beyond the minimum of the data? Default is `FALSE`.
`expand_max`	Should largest break be extended beyond the maximum of the data? Default is `FALSE`.

Value

A numeric vector of break-points.

Examples

```
library(cheapr)

set.seed(123)
ages <- sample(0:80, 100, TRUE)

get_breaks(ages, n = 10)

## To get the same break-points that `cut()` produces, we can do this..

signif(
  get_breaks(ages, n = 5, pretty = FALSE,
             expand_min = TRUE, expand_max = TRUE), 3
)
levels(cut(ages, 5))
```

---

is_na *Efficient functions for dealing with missing values.*

Description

`is_na()` is a parallelised alternative to `is.na()`.
`num_na(x)` is a faster and more efficient `sum(is.na(x))`.
`which_na(x)` is a more efficient `which(is.na(x))`
`which_not_na(x)` is a more efficient `which(!is.na(x))`
`row_na_counts(x)` is a more efficient `rowSums(is.na(x))`
`row_all_na()` returns a logical vector indicating which rows are empty and have only `NA` values.
`row_any_na()` returns a logical vector indicating which rows have at least 1 `NA` value.
The `col_` variants are the same, but operate by-column.

Usage

```
is_na(x)

## Default S3 method:
is_na(x)

## S3 method for class 'POSIXlt'
is_na(x)

## S3 method for class 'vctrs_rcrd'
is_na(x)
```

```
## S3 method for class 'data.frame'
is_na(x)

num_na(x, recursive = TRUE)

which_na(x)

which_not_na(x)

any_na(x, recursive = TRUE)

all_na(x, recursive = TRUE)

row_na_counts(x, names = FALSE)

col_na_counts(x, names = FALSE)

row_all_na(x, names = FALSE)

col_all_na(x, names = FALSE)

row_any_na(x, names = FALSE)

col_any_na(x, names = FALSE)
```

Arguments

`x`	A vector, list, data frame or matrix.
`recursive`	Should the function be applied recursively to lists? The default is `TRUE`. Setting this to `TRUE` is actually much cheaper because when `FALSE`, the other `NA` functions rely on calling `is_na()`, therefore allocating a vector. This is so that alternative objects with `is.na` methods can be supported.
`names`	Should row/col names be added?

Details

These functions are designed primarily for programmers, to increase the speed and memory-efficiency of `NA` handling.
Most of these functions can be parallelised through `options(cheapr.cores)`.

Common use-cases:

To replicate `complete.cases(x)`, use `!row_any_na(x)`.
To find rows with any empty values, use `which_(row_any_na(df))`.
To find empty rows use `which_(row_all_na(df))` or `which_na(df)`. To drop empty rows use `na_rm(df)` or `sset(df, which_(row_all_na(df), TRUE))`.

`is_na`**:**

`is_na` Is an S3 generic function. It will internally fall back on using `is.na` if it can't find a suitable method. Alternatively you can write your own `is_na` method. For example there is a method for `vctrs_rcrd` objects that simply converts it to a data frame and then calls `row_all_na()`. There is also a `POSIXlt` method for `is_na` that is much faster than `is.na`.

Lists:

When `x` is a list, `num_na`, `any_na` and `all_na` will recursively search the list for `NA` values. If `recursive = F` then `is_na()` is used to find `NA` values.
`is_na` differs to `is.na` in 2 ways:

- List elements are counted as `NA` if either that value is `NA`, or if it's a list, then all values of that list are `NA`.
- When called on a data frame, it returns `TRUE` for empty rows that contain only `NA` values.

Value

Number or location of `NA` values.

Examples

```
library(cheapr)
library(bench)

x <- 1:10
x[c(1, 5, 10)] <- NA
num_na(x)
which_na(x)
which_not_na(x)

row_nas <- row_na_counts(airquality, names = TRUE)
col_nas <- col_na_counts(airquality, names = TRUE)
row_nas
col_nas

df <- sset(airquality, j = 1:2)

# Number of NAs in data
num_na(df)
# Which rows are empty?
row_na <- row_all_na(df)
sset(df, row_na)

# Removing the empty rows
sset(df, which_(row_na, invert = TRUE))
# Or
na_rm(df)
# Or
sset(df, row_na_counts(df) < ncol(df))
```

`lag_` *Lagged operations.*

Description

Fast lags and leads optionally using dynamic vectorised lags, ordering and run lengths.

Usage

```
lag_(x, n = 1L, fill = NULL, set = FALSE, recursive = TRUE)

lag2_(
  x,
  n = 1L,
  order = NULL,
  run_lengths = NULL,
  fill = NULL,
  recursive = TRUE
)
```

Arguments

`x`	A vector or data frame.
`n`	Number of lags. Negative values are accepted. `lag2_` accepts a vector of dynamic lags and leads which gets recycled to the length of x.
`fill`	Value used to fill first n values. Default is `NA`.
`set`	Should x be updated by reference? If `TRUE` no copy is made and x is updated in place. The default is `FALSE`.
`recursive`	Should list elements be lagged as well? If `TRUE`, this is useful for data frames and will return row lags. If `FALSE` this will return a plain lagged list.
`order`	Optionally specify an ordering with which to apply the lags. This is useful for example when applying lags chronologically using an unsorted time variable.
`run_lengths`	Optional integer vector of run lengths that defines the size of each lag run. For example, supplying `c(5, 5)` applies lags to the first 5 elements and then essentially resets the bounds and applies lags to the next 5 elements as if they were an entirely separate and standalone vector. This is particularly useful in conjunction with the `order` argument to perform a by-group lag. See the examples for details.

Details

For most applications, it is more efficient and recommended to use `lag_()`. For anything that requires dynamic lags, lag by order of another variable, or by-group lags, one can use `lag2_()`.
To do cyclic lags, see the examples below for an implementation.

`lag2_`**:**

`lag2_` is a generalised form of `lag_` that by default performs simple lags and leads.
It has 3 additional features but does not support updating by reference or long vectors.

These extra features include:

- `n` - This shares the same name as the `n` argument in `lag_` for consistency. The difference is that `lag_` accepts a lag vector of length 1 whereas this accepts a vector of dynamic lags allowing for flexible combinations of variable sized lags and leads. These are recycled to the length of the data and will always align with the data, meaning that if you supply a custom `order` argument, this ordering is applied both to `x` and the recycled lag vector `n` simultaneously.
- `order` - Apply lags in any order you wish. This can be useful for reverse order lags, lags against unsorted time variables, and by-group lags.
- `run_lengths` - Specify the size of individual lag runs. For example, if you specify `run_lengths = c(3, 4, 2)`, this will apply your lags to the first 3 elements and then reset, applying lags to the next 4 elements, to reset again and apply lags to the final 2 elements. Each time the reset occurs, it treats each run length sized 'chunk' as a unique and separate vector. See the examples for a showcase.

Table of differences between `lag_` and `lag2_`:

Description	`lag_`	`lag2_`
Lags	Yes	Yes
Leads	Yes	Yes
Long vector support	Yes	No
Lag by reference	Yes	No
Dynamic vectorised lags	No	Yes
Data frame row lags	Yes	Yes
Alternative order lags	No	Yes

Value

A lagged object the same size as x.

Examples

```
library(cheapr)
library(bench)

# A use-case for data.table
# Adding 0 because can't update ALTREP by reference
df <- data.frame(x = 1:10^5 + 0L)

# Normal data frame lag
sset(lag_(df), 1:10)

# Lag these behind by 3 rows
sset(lag_(df, 3, set = TRUE), 1:10)
```

```
df$x[1:10] # x variable was updated by reference!

# The above can be used naturally in data.table to lag data
# without any copies

# To perform regular R row lags, just make sure set is `FALSE`

sset(lag_(as.data.frame(EuStockMarkets), 5), 1:10)

# lag2_ is a generalised version of lag_ that allows
# for much more complex lags

x <- 1:10

# lag every 2nd element
lag2_(x, n = c(1, 0)) # lag vector is recycled

# Explicit Lag(3) using a vector of lags
lags <- lag_sequence(length(x), 3, partial = FALSE)
lag2_(x, n = lags)

# Alternating lags and leads
lag2_(x, c(1, -1))

# Lag only the 3rd element
lags <- integer(length(x))
lags[3] <- 1L
lag2_(x, lags)

# lag in descending order (same as a lead)

lag2_(x, order = 10:1)

# lag that resets after index 5
lag2_(x, run_lengths = c(5, 5))

# lag with a time index
years <- sample(2011:2020)
lag2_(x, order = order(years))

# Example of how to do a cyclical lag
n <- length(x)

# When k >= 0
k <- min(3, n)
lag2_(x, c(rep(-n + k, k), rep(k, n - k)))
# When k < 0
k <- max(-3, -n)
lag2_(x, c(rep(k, n + k), rep(n + k, -k)))

# As it turns out, we can do a grouped lag
# by supplying group sizes as run lengths and group order as the order
```

```
set.seed(45)
g <- sample(c("a", "b"), 10, TRUE)

# NOTE: collapse::flag will not work unless g is already sorted!
# This is not an issue with lag2_()
collapse::flag(x, g = g)
lag2_(x, order = order(g), run_lengths = collapse::GRP(g)$group.sizes)

# For production code, we can of course make
# this more optimised by using collapse::radixorderv()
# Which calculates the order and group sizes all at once

o <- collapse::radixorderv(g, group.sizes = TRUE)
lag2_(x, order = o, run_lengths = attr(o, "group.sizes"))

# Let's finally wrap this up in a nice grouped-lag function

grouped_lag <- function(x, n = 1, g = integer(length(x))){
  o <- collapse::radixorderv(g, group.sizes = TRUE, sort = FALSE)
  lag2_(x, n, order = o, run_lengths = attr(o, "group.sizes"))
}

# And voila!
grouped_lag(x, g = g)

# A method to extract this information from dplyr

## We can actually get this information easily from a `grouped_df` object
## Uncomment the below code to run the implementation
# library(dplyr)
# library(timeplyr)
# eu_stock <- EuStockMarkets |>
#   ts_as_tibble() |>
#   group_by(stock_index = group)
# groups <- group_data(eu_stock) # Group information
# group_order <- unlist(groups$.rows) # Order of groups
# group_sizes <- lengths_(groups$.rows) # Group sizes
#
# # by-stock index lag
# lag2_(eu_stock$value, order = group_order, run_lengths = group_sizes)
#
# # Verifying this output is correct
# eu_stock |>
#   ungroup() |>
#   mutate(lag1 = lag_(value), .by = stock_index) |>
#   mutate(lag2 = lag2_(value, order = group_order, run_lengths = group_sizes)) |>
#   summarise(lags_are_equal = identical(lag1, lag2))

# Let's compare this to data.table

library(data.table)
default_threads <- getDTthreads()
setDTthreads(1)
```

```
dt <- data.table(x = 1:10^5,
                 g = sample.int(10^4, 10^5, TRUE))

bench::mark(dt[, y := shift(x), by = g][][["y"]],
            grouped_lag(dt$x, g = dt$g),
            iterations = 10)
setDTthreads(default_threads)
```

lengths_ *List utilities*

Description

Functions to help work with lists.

Usage

```
lengths_(x, names = FALSE)

unlisted_length(x)

new_list(length = 0L, default = NULL)
```

Arguments

`x`	A list.
`names`	Should names of list elements be added? Default is `FALSE`.
`length`	Length of list.
`default`	Default value for each list element.

Value

`lengths_()` returns the list lengths.
`unlisted_length()` is an alternative to `length(unlist(x))`.
`new_list()` is like `vector("list", length)` but also allows you to specify a default value for each list element. This can be useful for initialising with a catch-all value so that when you unlist you're guaranteed a list of length >= to the specified length.

Examples

```
library(cheapr)
l <- list(1:10,
          NULL,
          list(integer(), NA_integer_, 2:10))

lengths_(l) # Faster lengths()
unlisted_length(l) # length of vector if we unlist
paste0("length: ", length(print(unlist(l))))
```

```
unlisted_length(l) - na_count(l) # Number of non-NA elements

# We can create and initialise a new list with a default value
l <- new_list(20, 0L)
l[1:5]
# This works well with vctrs_list_of objects
```

`named_list` *Turn dot-dot-dot (...) into a named list*

Description

A fast and useful function for always returning a named list from `...`

Usage

```
named_list(..., .keep_null = TRUE)
```

Arguments

`...`	Key-value pairs.
`.keep_null`	Should `NULL` entries be kept? Default is `TRUE`.

Value

A named list.

`new_df` *Fast data frame constructor*

Description

Fast data frame constructor

Usage

```
new_df(..., .nrows = NULL, .recycle = FALSE, .name_repair = FALSE)
```

Arguments

`...`	Key-value pairs.
`.nrows`	`integer(1)` (Optional) number of rows. Commonly used to initialise a 0-column data frame with rows.
`.recycle`	`logical(1)` Should arguments be recycled? Default is `FALSE`.
`.name_repair`	`logical(1)` Should duplicate names be made unique? Default is `FALSE`.

Value

A `data.frame`

`overview` *An alternative to* `summary()` *inspired by the skimr package*

Description

A cheaper `summary()` function, designed for larger data.

Usage

```
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## Default S3 method:
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'logical'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'integer'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'numeric'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'integer64'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'character'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'factor'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'Date'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'POSIXt'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'ts'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))

## S3 method for class 'zoo'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))
```

```
## S3 method for class 'data.frame'
overview(x, hist = FALSE, digits = getOption("cheapr.digits", 2))
```

Arguments

`x`	A vector or data frame.
`hist`	Should in-line histograms be returned? Default is `FALSE`.
`digits`	How many decimal places should the summary statistics be printed as? Default is 2.

Details

No rounding of statistics is done except in printing which can be controlled either through the `digits` argument in `overview()`, or by setting the option `options(cheapr.digits)`.
To access the underlying data, for example the numeric summary, just use `$numeric`, e.g. `overview(rnorm(30))$numeric`.

Value

An object of class "overview". Under the hood this is just a list of data frames. Key summary statistics are reported in each data frame.

Examples

```
library(cheapr)
overview(iris)

# With histograms
overview(airquality, hist = TRUE)

# Round to 0 decimal places
overview(airquality, digits = 0)

# We can set an option for all overviews
options(cheapr.digits = 1)
overview(rnorm(100))
options(cheapr.digits = 2) # The default
```

`recycle` *Recycle objects to a common size*

Description

A convenience function to recycle R objects to either a common or specified size.

Usage

```
recycle(..., length = NULL)
```

Arguments

`...`	Objects to recycle.
`length`	Optional length to recycle objects to.

Details

Data frames are recycled by recycling their rows.
`recycle()` is optimised to only recycle objects that need recycling.
`NULL` objects are ignored and not recycled or returned.

Value

A list of recycled R objects.

Examples

```
library(cheapr)

recycle(Sys.Date(), 1:10)

# Any vectors of zero-length are all recycled to zero-length
recycle(integer(), 1:10)

# Data frame rows are recycled
recycle(sset(iris, 1:3), length = 3 * 3)

# To recycle list items, use `do.call()`
my_list <- list(from = 1L, to = 10L, by = seq(0.1, 1, 0.1))
do.call(recycle, my_list)
```

`sequence_` *Utilities for creating many sequences*

Description

`sequence_` is an extension to sequence which accepts decimal number increments.
`seq_id` can be paired with `sequence_` to group individual sequences.
`seq_` is a vectorised version of seq.
`window_sequence` creates a vector of window sizes for rolling calculations.
`lag_sequence` creates a vector of lags for rolling calculations.
`lead_sequence` creates a vector of leads for rolling calculations.

Usage

```
sequence_(size, from = 1L, by = 1L, add_id = FALSE)

seq_id(size)

seq_(from = 1L, to = 1L, by = 1L, add_id = FALSE)

seq_size(from, to, by = 1L)

window_sequence(size, k, partial = TRUE, ascending = TRUE, add_id = FALSE)

lag_sequence(size, k, partial = TRUE, add_id = FALSE)

lead_sequence(size, k, partial = TRUE, add_id = FALSE)
```

Arguments

`size`	Vector of sequence lengths.
`from`	Start of sequence(s).
`by`	Unit increment of sequence(s).
`add_id`	Should the ID numbers of the sequences be added as names? Default is `FALSE`.
`to`	End of sequence(s).
`k`	Window/lag size.
`partial`	Should partial windows/lags be returned? Default is `TRUE`.
`ascending`	Should window sequence be ascending? Default is `TRUE`.

Details

`sequence_()` works in the same way as `sequence()` but can accept non-integer `by` values. It also recycles `from` and `to`, in the same way as `sequence()`.
If any of the sequences contain values > `.Machine$integer.max`, then the result will always be a double vector.

`from` can be also be a date, date-time, or any object that supports addition and multiplication.

`seq_()` is a vectorised version of `seq()` that strictly accepts only the arguments `from`, `to` and `by`.

Value

A vector of length `sum(size)` except for `seq_` which returns a vector of size `sum((to - from) / (by + 1))`

Examples

```
library(cheapr)
sequence(1:3)
sequence_(1:3)
```

```
sequence(1:3, by = 0.1)
sequence_(1:3, by = 0.1)

# Add IDs to the sequences
sequence_(1:3, by = 0.1, add_id = TRUE)
# Turn this quickly into a data frame
enframe_(sequence_(1:3, by = 0.1, add_id = TRUE))

sequence(c(3, 2), by = c(-0.1, 0.1))
sequence_(c(3, 2), by = c(-0.1, 0.1))


# Vectorised version of seq()
seq_(1, 10, by = c(1, 0.5))
# Same as below
c(seq(1, 10, 1), seq(1, 10, 0.5))

# Programmers may use seq_size() to determine final sequence lengths

sizes <- seq_size(1, 10, by = c(1, 0.5))
print(paste(c("sequence sizes: (", sizes, ") total size:", sum(sizes)),
            collapse = " "))

# We can group sequences using seq_id

from <- Sys.Date()
to <- from + 10
by <- c(1, 2, 3)
x <- seq_(from, to, by, add_id = TRUE)
class(x) <- "Date"
x

# Utilities for rolling calculations

window_sequence(c(3, 5), 3)
window_sequence(c(3, 5), 3, partial = FALSE)
window_sequence(c(3, 5), 3, partial = TRUE, ascending = FALSE)
# One can for example use these in data.table::frollsum
```

setdiff_ *Extra utilities*

Description

Extra utilities

Usage

```
setdiff_(x, y, dups = TRUE)

intersect_(x, y, dups = TRUE)

cut_numeric(
  x,
  breaks,
  labels = NULL,
  include.lowest = FALSE,
  right = TRUE,
  dig.lab = 3L,
  ordered_result = FALSE,
  ...
)

## S3 method for class 'integer64'
cut(x, ...)

x %in_% table

x %!in_% table

enframe_(x, name = "name", value = "value")

deframe_(x)

sample_(x, size = vector_length(x), replace = FALSE, prob = NULL)

val_insert(x, value, n = NULL, prop = NULL)

na_insert(x, n = NULL, prop = NULL)

vector_length(x)
```

Arguments

x	A vector or data frame.
y	A vector or data frame.
dups	Should duplicates be kept? Default is TRUE.
breaks	See ?cut.
labels	See ?cut.
include.lowest	See ?cut.
right	See ?cut.
dig.lab	See ?cut.
ordered_result	See ?cut.

`...`	Further arguments passed onto `cut` or `set.seed`.
`table`	See `?collapse::fmatch`
`name`	The column name to assign the names of a vector.
`value`	The column name to assign the values of a vector.
`size`	See `?sample`.
`replace`	See `?sample`.
`prob`	See `?sample`.
`n`	Number of scalar values (or `NA`) to insert randomly into your vector.
`prop`	Proportion of scalar values (or `NA`) values to insert randomly into your vector.

Details

`intersect_()` and `setdiff_()` are faster and more efficient alternatives to `intersect()` and `setdiff()` respectively.
`enframe_()` and `deframe_()` are faster alternatives to `tibble::enframe()` and `tibble::deframe()` respectively.
`cut_numeric()` is a faster and more efficient alternative to `cut.default()`.

Value

`enframe()_` converts a vector to a data frame.
`deframe()_` converts a 1-2 column data frame to a vector.
`intersect_()` returns a vector of common values between `x` and `y`.
`setdiff_()` returns a vector of values in `x` but not `y`.
`cut_numeric()` places values of a numeric vector into buckets, defined through the `breaks` argument and returns a factor unless `labels = FALSE`, in which case an integer vector of break indices is returned.
`%in_%` and `%!in_%` both return a logical vector signifying if the values of `x` exist or don't exist in `table` respectively.
`sample_()` is an alternative to `sample()` that natively samples data frame rows through `sset()`. It also does not have a special case when `length(x)` is 1.
`val_insert` inserts scalar values randomly into your vector. Useful for replacing lots of data with a single value.
`na_insert` inserts `NA` values randomly into your vector. Useful for generating missing data.
`vector_length` behaves mostly like `NROW()` except for matrices in which it matches `length()`.

`set_abs` *Math operations by reference - **Experimental***

Description

These functions transform your variable by reference, with no copies being made. It is advisable to only use these if you know what you are doing.

Usage

```
set_abs(x)

set_floor(x)

set_ceiling(x)

set_trunc(x)

set_exp(x)

set_sqrt(x)

set_change_sign(x)

set_round(x, digits = 0)

set_log(x, base = exp(1))

set_pow(x, y)

set_add(x, y)

set_subtract(x, y)

set_multiply(x, y)

set_divide(x, y)
```

Arguments

`x`	A numeric vector.
`digits`	Number of digits to round to.
`base`	Logarithm base.
`y`	A numeric vector.

Details

These functions are particularly useful for situations where you have made a copy and then wish to perform further operations without creating more copies.
`NA` and `NaN` values are ignored though in some instances `NaN` values may be replaced with `NA`. These functions will **not work** on **any** classed objects, meaning they only work on standard integer and numeric vectors and matrices.

When a copy has to be made:

A copy is only made in certain instances, e.g. when passing an integer vector to `set_log()`. A warning will always be thrown in this instance alerting the user to assign the output to an object

because `x` has not been updated by reference.
To ensure consistent and expected outputs, always assign the output to the same object,
e.g. `x` <- `set_log(x)` (**do this**)
`set_log(x)` (**don't do this**)
`x2` <- `set_log(x)` (Don't do this either)

No copy is made here unless `x` is an integer vector.

Value

The exact same object with no copy made, just transformed.

Examples

```
library(cheapr)
library(bench)

x <- rnorm(2e05)
options(cheapr.cores = 2)
mark(
  base = exp(log(abs(x))),
  cheapr = set_exp(set_log(set_abs(x)))
)
options(cheapr.cores = 1)
```

`sset` *Cheaper subset*

Description

Cheaper alternative to `[` that consistently subsets data frame rows, always returning a data frame. There are explicit methods for enhanced data frames like tibbles, data.tables and sf.

Usage

```
sset(x, ...)

## S3 method for class 'Date'
sset(x, i, ...)

## S3 method for class 'POSIXct'
sset(x, i, ...)

## S3 method for class 'factor'
sset(x, i, ...)

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

```
sset(x, i, j, ...)

## S3 method for class 'tbl_df'
sset(x, i, j, ...)

## S3 method for class 'POSIXlt'
sset(x, i, j, ...)

## S3 method for class 'data.table'
sset(x, i, j, ...)

## S3 method for class 'sf'
sset(x, i, j, ...)
```

Arguments

`x`	Vector or data frame.
`...`	Further parameters passed to `[`.
`i`	A logical or vector of indices.
`j`	Column indices, names or logical vector.

Details

`sset` is an S3 generic. You can either write methods for `sset` or `[`.
`sset` will fall back on using `[` when no suitable method is found.

To get into more detail, using `sset()` on a data frame, a new list is always allocated through `new_list()`.

Difference to base R:
When `i` is a logical vector, it is passed directly to `which_()`.
This means that `NA` values are ignored and this also means that `i` is not recycled, so it is good practice to make sure the logical vector matches the length of x. To return `NA` values, use `sset(x, NA_integer_)`.

ALTREP range subsetting:
When `i` is an ALTREP compact sequence which can be commonly created using e.g. `1:10` or using `seq_len`, `seq_along` and `seq.int`, `sset` internally uses a range-based subsetting method which is faster and doesn't allocate `i` into memory.

Value

A new vector, data frame, list, matrix or other R object.

Examples

```
library(cheapr)
library(bench)
```

```
# Selecting columns
sset(airquality, j = "Temp")
sset(airquality, j = 1:2)

# Selecting rows
sset(iris, 1:5)

# Rows and columns
sset(iris, 1:5, 1:5)
sset(iris, iris$Sepal.Length > 7, c("Species", "Sepal.Length"))

# Comparison against base
x <- rnorm(10^4)

mark(x[1:10^3], sset(x, 1:10^3))
mark(x[x > 0], sset(x, x > 0))

df <- data.frame(x = x)

mark(df[df$x > 0, , drop = FALSE],
     sset(df, df$x > 0),
     check = FALSE) # Row names are different


## EXTRA: An easy way to incorporate cheapr into dplyr's filter()
# cheapr_filter <- function(.data, ..., .by = NULL, .preserve = FALSE){
#   filter_df <- .data |>
#     dplyr::mutate(..., .by = {{ .by }}, .keep = "none")
#   groups <- dplyr::group_vars(filter_df)
#   filter_df <- cheapr::sset(filter_df, j = setdiff(names(filter_df), groups))
#   n_filters <- ncol(filter_df)
#   if (n_filters < 1){
#     .data
#   } else {
#     dplyr::dplyr_row_slice(.data, cheapr::which_(Reduce(`&`, filter_df)),
#                            preserve = .preserve)
#   }
# }
```

`val_count` *Efficient functions for counting, finding, replacing and removing scalars*

Description

These are primarily intended as very fast scalar-based functions for developers. They are particularly useful for working with `NA` values in a fast and efficient manner.

Usage

```
val_count(x, value, recursive = TRUE)

count_val(x, value, recursive = TRUE)

val_find(x, value, invert = FALSE)

which_val(x, value, invert = FALSE)

val_replace(x, value, replace, recursive = TRUE)

na_replace(x, replace, recursive = TRUE)

val_rm(x, value)

na_count(x, recursive = TRUE)

na_find(x, invert = FALSE)

na_rm(x)
```

Arguments

`x`	A vector, list, data frame or matrix.
`value`	A scalar value to count, find, replace or remove.
`recursive`	Should values in a list be counted or replaced recursively? Default is `TRUE` and very useful for data frames.
`invert`	Should `which_val` find locations of everything except specified value? Default is `FALSE`.
`replace`	Replacement scalar value.

Details

The `val_` functions allow you to very efficiently work with scalars, i.e length 1 vectors. Many common common operations like counting the occurrence of `NA` or zeros, e.g. `sum(x == 0)` or `sum(is.na(x))` can be replaced more efficiently with `val_count(x, 0)` and `na_count(x)` respectively.

At the moment these functions only work for integer, double and character vectors with the exception of the `NA` functions. They are intended mainly for developers who wish to write cheaper code and reduce expensive vector operations.

- `val_count()` - Counts occurrences of a value
- `val_find()` Finds locations (indices) of a value
- `val_replace()` - Replaces value with another value
- `val_rm()` - Removes occurrences of value from an object

There are `NA` equivalent convenience functions.

- `na_count()` == `val_count(x, NA)`
- `na_find()` == `val_find(x, NA)`
- `na_replace()` == `val_replace(x, NA)`
- `na_rm()` == `val_rm(x, NA)`

`val_count()` and `val_replace()` can work recursively. For example, when applied to a data frame, `na_replace` will replace `NA` values across the entire data frame with the specified replacement value.

In 'cheapr' function-naming conventions have not been consistent but going forward all scalar functions (including the `NA` convenience functions) will be prefixed with 'val_' and 'na_' respectively. Functions named with the older naming scheme like `which_na` may be removed at some point in the future.

Value

`val_count()` returns the number of times a scalar value appears in a vector or list.
`val_find()` returns the index locations of that scalar value.
`val_replace()` replaces a specified scalar value with a replacement scalar value. If no instances of said value are found then the input x is returned as is.
`na_replace()` is a convenience function equivalent to `val_replace(x, NA, ...)`.
`val_rm()` removes all instances of a specified scalar value. If no instances are found, the original input x is returned as is.

`which_` *Memory-efficient alternative to* `which()`

Description

Exactly the same as `which()` but more memory efficient.

Usage

```
which_(x, invert = FALSE)
```

Arguments

`x`	A logical vector.
`invert`	If `TRUE`, indices of values that are not `TRUE` are returned (including `NA`). If `FALSE` (the default), only `TRUE` indices are returned.

Details

This implementation is similar in speed to `which()` but usually more memory efficient.

Value

An unnamed integer vector.

Examples

```
library(cheapr)
library(bench)
x <- sample(c(TRUE, FALSE), 1e05, TRUE)
x[sample.int(1e05, round(1e05/3))] <- NA

mark(which_(TRUE), which(TRUE))
mark(which_(FALSE), which(FALSE))
mark(which_(logical()), which(logical()))
mark(which_(x), which(x), iterations = 20)
mark(base = which(is.na(match(x, TRUE))),
     collapse = collapse::whichv(x, TRUE, invert = TRUE),
     cheapr = which_(x, invert = TRUE),
     iterations = 20)
```

Index