Package 'cheapr'

May 8, 2026

Title Simple Functions to Save Time and Memory

Version 1.5.1

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 (>= 4.1.0)

Imports collapse (>= 2.0.0)

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

LinkingTo cpp11

SystemRequirements C++17

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation yes

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

Repository CRAN

Date/Publication 2026-04-04 05:11:34 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

---

address *Memory address of R object*

Description

Memory address of R object

Usage

```
address(x)
```

Arguments

x	An R object.

Value

Memory address of R object.

---

as_discrete *Turn continuous data into discrete bins*

Description

This is a cheapr version of `cut.numeric()` which is more efficient and prioritises pretty-looking breaks by default through the use of `get_breaks()`. Out-of-bounds values can be included naturally through the `include_oob` argument. Left-closed (right-open) intervals are returned by default in contrast to cut's default right-closed intervals. Furthermore there is flexibility in formatting the interval bins, allowing the user to specify formatting functions and symbols for the interval close and open symbols.

Usage

```
as_discrete(x, ...)

## S3 method for class 'numeric'
as_discrete(
  x,
  breaks = if (left_closed) get_breaks(x) else rev_(-get_breaks(-x)),
  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 = ",",
  inf_label = NULL,
  ...
)

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

Arguments

x	A numeric vector.
...	Extra arguments passed onto methods.
breaks	Break-points. The default option creates pretty looking breaks. Unlike `cut()`, the `breaks` arg cannot be a number denoting the number of breaks you want. To generate breakpoints this way use `get_breaks()`.
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`. This is equivalent to `breaks = c(breaks, Inf)` or `breaks = c(-Inf, breaks)` when `left_closed = FALSE`. If `include_endpoint = TRUE`, the endpoint interval is prioritised before the out-of-bounds interval. This behaviour cannot be replicated easily with `cut()`. For example, these 2 expressions are not equivalent: `cut(10, c(9, 10, Inf), right = F, include.lowest = T) != as_discrete(10, c(9, 10), include_endpoint = T, include_oob = T)`
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.
inf_label	Label to use for intervals that include infinity. If left NULL the Unicode infinity symbol is used.

Value

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

See Also

bin get_breaks

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)

# To closely replicate `cut()` with `as_discrete()` we can use the following

cheapr_cut <- function(x, breaks, right = TRUE,
                       include.lowest = FALSE,
                       ordered.result = FALSE){
  if (length(breaks) == 1){
    breaks <- get_breaks(x, breaks, pretty = FALSE,
                         expand_min = FALSE, expand_max = FALSE)
    adj <- diff(range(breaks)) * 0.001
    breaks[1] <- breaks[1] - adj
    breaks[length(breaks)] <- breaks[length(breaks)] + adj
  }
  as_discrete(x, breaks, left_closed = !right,
              include_endpoint = include.lowest,
              ordered = ordered.result,
              intv_start_fun = function(x) formatC(x, digits = 3, width = 1),
              intv_end_fun = function(x) formatC(x, digits = 3, width = 1))
}

x <- rnorm(100)
cheapr_cut(x, 10)
identical(cut(x, 10), cheapr_cut(x, 10))
```

`attrs` *Add and remove attributes*

Description

Simple tools to add and remove attributes, both normally and in-place. To remove specific attributes, set those attributes to `NULL`.

Usage

```
attrs_modify(x, ..., .set = FALSE, .args = NULL)

attrs_add(x, ..., .set = FALSE, .args = NULL)

attrs_clear(x, .set = FALSE)

attrs_rm(x, .set = FALSE)
```

Arguments

`x`	Object to add/remove attributes.
`...`	Named attributes, e.g 'key = value'.

`.set`	Should attributes be added in-place without shallow-copying `x`? Default is `FALSE`.
`.args`	An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.

Value

The object `x` with attributes removed or added.

See Also

shallow_copy

---

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`. When `TRUE`, this essentially becomes `findInterval()`.
`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.

See Also

get_breaks as_discrete

`case` *A cheapr case-when and switch*

Description

`case` and `val_match` are cheaper alternatives to `dplyr::case_when` and `dplyr::case_match` respectively.

Usage

```
case(..., .default = NULL)

val_match(.x, ..., .default = NULL)
```

Arguments

`...`	Logical expressions or scalar values in the case of `val_match`.
`.default`	Catch-all value or vector.
`.x`	Vector used to switch values.

Details

`val_match()` is a very efficient special case of the `case()` function when all lhs expressions are scalars, i.e. length-1 vectors. RHS expressions can be vectors the same length as `.x`. The below 2 expressions are equivalent.

```
val_match(
  x,
  1 ~ "one",
  2 ~ "two",
  .default = "Unknown"
 )
case(
  x == 1 ~ "one",
  x == 2 ~ "two",
  .default = "Unknown"
 )
```

Value

A vector the same length as `.x` or same length as the first condition in the case of `case`, unless the condition length is smaller than the rhs, in which case the length of the rhs is used.

See Also

if_else_

`cast` *Fast casting/coercing of R objects*

Description

`cast_common()` is type-commutative, meaning the order of objects doesn't affect the outcome type.
`cast()` will attempt to cast `x` into an object similar to `archetype`.

Usage

```
cast(x, archetype)

cast_common(..., .args = NULL)

archetype(x)

archetype_common(..., .args = NULL)

r_type(x)

r_type_common(..., .args = NULL)
```

Arguments

`x`	A vector.
`archetype`	An archetype vector.
`...`	Vectors.
`.args`	An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.

Value

`cast()` will attempt to cast `x` into an object similar to `archetype`.
`cast_common()` coerces all supplied vectors into a common type between them.
`archetype()` returns the zero-length template/archetype of `x`.
`archetype_common()` returns the common zero-length template between all supplied vectors.
`r_type()` will return the internal cheapr-defined type of `x` as a character vector of length 1. This will usually match `class(x)` but not always.
`r_type_common()` returns the common type between all objects.

`cheapr_table` *Fast frequency tables - Still experimental*

Description

This is not a one-to-one copy of `base::table()` as some behaviours differ. It is more flexible as it accepts inputs such as data frames and `vctrs_rcrd` objects.

Usage

```
cheapr_table(
  ...,
  names = TRUE,
  order = FALSE,
  na_exclude = FALSE,
  classed = FALSE
)

counts(x, sort = is.factor(x))

table_(..., names = TRUE, order = FALSE, na_exclude = FALSE, classed = FALSE)
```

Arguments

`...`	>=1 objects that can be converted to a factor through `cheapr::factor_()`.
`names`	Should level names be kept? Default is `TRUE`.
`order`	Should result be ordered by level names? Default is `FALSE`.
`na_exclude`	Should `NA` values be excluded? Default is `FALSE`.
`classed`	Should a `table` object be returned? Default is `FALSE`
`x`	A vector.
`sort`	Should groups be sorted? Default is `FALSE`.

Details

`cheapr_table()` tries to match the behaviour of `table()` where possible. `counts()` is an alternative that returns a `data.frame` of unique keys and counts.

Value

A named integer vector if one object is supplied, otherwise an array.

copy *Copy R objects*

Description

shallow_copy() and deep_copy() are just wrappers to the R C API functions Rf_shallow_duplicate() and Rf_duplicate() respectively. semi_copy() is something in between whereby it fully copies the data but only shallow copies the attributes.

Usage

```
shallow_copy(x)

semi_copy(x)

deep_copy(x)
```

Arguments

x	An object to shallow, semi, or deep copy.

Details

Shallow duplicates are mainly useful for adding attributes to objects in-place as well assigning vectors to shallow copied lists in-place.

Deep copies are generally useful for ensuring an object is fully duplicated, including all attributes associated with it. Deep copies are generally expensive and should be used with care.

semi_copy() deep copies everything except the attributes. This is experimental but in theory should be much more efficient and generally preferred to deep_copy().

To summarise:

- shallow_copy - Shallow copies data and attributes
- semi_copy - Deep copies data and shallow copies attributes
- deep_copy - Deep copies both data and attributes

It is recommended to use these functions only if you know what you are doing.

Value

A shallow, semi or deep copied R object.

Examples

```
library(cheapr)
library(bench)
df <- new_df(x = sample.int(10^4))

# Note the memory allocation
mark(shallow_copy(df), iterations = 1)
mark(deep_copy(df), iterations = 1)

# In both cases the address of df changes

address(df);address(shallow_copy(df));address(deep_copy(df))

# When shallow-copying attributes are not duplicated

address(attr(df, "names"));address(attr(shallow_copy(df), "names"))

# They are when deep-copying

address(attr(df, "names"));address(attr(deep_copy(df), "names"))

# Adding an attribute in place with and without shallow copy
invisible(attrs_add(df, key = TRUE, .set = TRUE))
attr(df, "key")

# Remove attribute in-place
invisible(attrs_add(df, key = NULL, .set = TRUE))

# With shallow copy
invisible(attrs_add(shallow_copy(df), key = TRUE, .set = TRUE))

# 'key' attr was only added to the shallow copy, and not the original df
attr(df, "key")
```

---

cpp_rebuild *Low-level attribute re-constructor*

Description

Low-level attribute re-constructor

Usage

```
cpp_rebuild(target, source, target_attr_names, source_attr_names, shallow_copy)
```

Arguments

target	Target object you wish to rebuild attributes on.

`source` Source object to copy attributes from.

`target_attr_names`
`[character(n)]` Names of target attributes to keep.

`source_attr_names`
`[character(n)]` Names of source attributes to copy onto target.

`shallow_copy` `[logical(1)]` Should target be shallow copied before re-building? If `FALSE` attributes are added in-place.

Details

`cpp_rebuild()` is mostly a convenience function to help with choosing exactly which attributes to copy onto the target object. `rebuild()` is a related generic function with rebuild methods for common objects (currently only `tbl_df`, `data.frame` and `data.table`). For examples of further rebuild methods, see the fastplyr package.

To modify attributes yourself you can of course use base R attribute functions like `attr()` and `attributes()` or cheapr's more convenient `attrs_modify`.

Value

An object similar to `source`.

See Also

rebuild attrs_modify

---

c_ *A cheapr version of* `c()`

Description

cheapr's version of `c()`. It is quite a bit faster for atomic vectors and combines data frame rows instead of cols.

Usage

```
c_(..., .args = NULL)

cheapr_c(..., .args = NULL)
```

Arguments

`...` Objects to combine.

`.args` An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.

Value

Combined objects.

Examples

```
library(cheapr)

# Combine just like `c()`
c_(1, 2, 3:5)

# It combines rows by default instead of cols
c_(new_df(x = 1:3), new_df(x = 4:10))

# If you have a list of objects you want to combine
# use `.args` instead of `do.call` as it's more efficient

list_of_objs <- rep_(list(0), 10^4)

bench::mark(
  do.call(c, list_of_objs),
  do.call(c_, list_of_objs),
  c_(.args = list_of_objs) # Fastest
)
```

---

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()` adds the specified levels onto the existing levels, `levels_rm()` removes the specified levels, `levels_add_na()` which adds an explicit `NA` level, `levels_drop_na()` which drops the `NA` level, `levels_drop()` which drops unused factor levels, `levels_rename()` for renaming levels, `levels_lump()` which returns top n levels and lumps all others into the same category,
`levels_count()` which returns the counts of each level, 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)

levels_rm(x, levels)

levels_add(x, levels, where = c("last", "first"))

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

levels_drop_na(x)

levels_drop(x)

levels_reorder(x, order_by, decreasing = FALSE)

levels_rename(x, ..., .fun = NULL)

levels_lump(
  x,
  n,
  prop,
  other_category = "Other",
  ties = c("min", "average", "first", "last", "random", "max")
)

levels_count(x)
```

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?
where	Where should NA level be placed? Either first or last.
name	Name of NA level.
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.

...	Key-value pairs where the key is the new name and value is the name to replace that with the new name. For example `levels_rename(x, new = old)` replaces the level "old" with the level "new".
.fun	Renaming function applied to each level.
n	Top n number of levels to calculate.
prop	Top proportion of levels to calculate. This is a proportion of the total unique levels in x.
other_category	Name of 'other' category.
ties	Ties method to use. See ?rank.

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.

`levels_lump()` is a cheaper version of `forcats::lump_n()` but returns levels in order of highest frequency to lowest. This can be very useful for plotting.

Value

A `factor` or `character` in the case of `levels_used` and `levels_unused`. `levels_count` returns a data frame of counts and proportions for each level.

Examples

```
library(cheapr)

x <- factor_(sample(letters[sample.int(26, 10)], 100, TRUE), levels = letters)
x
# Used/unused levels

levels_used(x)
levels_unused(x)

# Drop unused levels
levels_drop(x)

# Top 3 letters by by frequency
lumped_letters <- levels_lump(x, 3)
levels_count(lumped_letters)

# To remove the "other" category, use `levels_rm()`

levels_count(levels_rm(lumped_letters, "Other"))
```

```
# We can use levels_lump to create a generic top n function for non-factors too

get_top_n <- function(x, n){
  f <- levels_lump(factor_(x, order = FALSE), n = n)
  levels_count(f)
}

get_top_n(x, 3)

# A neat way to order the levels of a factor by frequency
# is the following:

levels(levels_lump(x, prop = 1)) # Highest to lowest
levels(levels_lump(x, prop = -1)) # Lowest to highest
```

---

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 = 10, ...)

## Default S3 method:
get_breaks(x, n = 10, ...)

## S3 method for class 'numeric'
get_breaks(
  x,
  n = 10,
  pretty = TRUE,
  expand_min = FALSE,
  expand_max = pretty,
  ...
)

## S3 method for class 'integer64'
get_breaks(x, n = 10, ...)
```

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. If FALSE bin-widths will be calculated as diff(range(x)) / n.
expand_min	Should smallest break be extended beyond the minimum of the data? Default is FALSE. If TRUE then min(get_breaks(x)) is ensured to be less than min(x).
expand_max	Should largest break be extended beyond the maximum of the data? Default is TRUE. If TRUE then max(get_breaks(x)) is ensured to be greater than max(x).

Value

A numeric vector of break-points.

See Also

bin as_discrete

Examples

```
library(cheapr)

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

# Pretty
get_breaks(ages, n = 10)
# Not-pretty
# bin-width is diff(range(ages)) / n_breaks
get_breaks(ages, n = 10, pretty = FALSE)

# `get_breaks()` is left-biased in a sense, meaning that
# the first break is always <= `min(x)` but the last break
# may be < `max(x)`

# To get right-biased breaks we can use a helper like so..

right_breaks <- function(x, ...){
  -get_breaks(-x, ...)
}

get_breaks(4:24, 10)
right_breaks(4:24, 10)

# Use `rev()` to ensure they are in ascending order
rev(right_breaks(4:24, 10))
```

get_max_threads *Get and set the number of OpenMP threads to be used in cheapr*

Description

The default number of threads in cheapr is 2 (if your system has at least 2 available threads). You can change the number of threads via `set_threads()`. To see the number of threads currently being used, run `get_threads()`. To see the maximum number of threads your machine can use, run `get_max_threads()`

Usage

```
get_max_threads()

set_threads(n)

get_threads()
```

Arguments

`n`	`[integer(1)]` - Number of threads to use.

Value

`get_max_threads()` returns the max number of threads you can use.
`get_threads()` returns the number of threads that are being used.
`set_threads()` invisibly sets the number of threads to be used.

if_else *Cheaper version of* `ifelse()`

Description

Cheaper version of `ifelse()`

Usage

```
if_else_(condition, true, false, na = NULL)

cheapr_if_else(condition, true, false, na = NULL)
```

Arguments

`condition`	logical A condition which will be used to evaluate the if else operation.
`true`	Value(s) to replace `TRUE` instances.
`false`	Value(s) to replace `FALSE` instances.
`na`	Catch-all value(s) to replace all other instances, where `is.na(condition)`.

Value

A vector the same length as condition, using a common type between `true`, `false` and `na`.

See Also

case val_match

`int_sign` *A fast and integer-based* `sign()`

Description

A fast and integer-based `sign()`

Usage

```
int_sign(x)
```

Arguments

`x`	Integer or double vector.

Value

An integer vector denoting the sign, -1 for negatives, 1 for positives and 0 for when `x == 0`.

`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))
```

`is_whole_number` *Very fast check that numeric vector consists only of whole numbers*

Description

Very fast check that numeric vector consists only of whole numbers

Usage

```
is_whole_number(x, tol = sqrt(.Machine$double.eps), na.rm = TRUE)
```

Arguments

`x`	`[numeric(n)]` - A numeric vector.
`tol`	`[numeric(1)]` - Tolerance.
`na.rm`	`[logical(1)]` - Should `NA` values be ignored? Default is `TRUE`.

Details

`is_whole_number()` will return `NA` when these 3 conditions are met:

- `na.rm` is `FALSE`
- `x` contains at least 1 `NA` value
- `x` contains only a mix of whole numbers and/or `NA` values. If any values are not whole numbers then we can return `FALSE` even with the presence of `NA` values.

If `x` is not numeric then `is_whole_number()` always returns `FALSE`.

Value

`TRUE`, `FALSE`, or `NA` (see Details)

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)
```

`list_lengths` *List utilities*

Description

Functions to help work with lists.

Usage

```
list_lengths(x, names = FALSE)

lengths_(x, names = FALSE)

unlisted_length(x)

new_list(length = 0L, default = NULL)

list_assign(x, values)

list_modify(x, values)

list_combine(..., .args = NULL)

list_drop_null(x)
```

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.
`values`	A named list
`...`	Objects to combine into a list.
`.args`	An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.

Value

`list_lengths()` returns the list lengths.
`unlisted_length()` is a fast 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.

`list_assign()` is vectorised version of `[[<-` that concatenates `values` to `x` or modifies `x` where the names match. Can be useful for modifying data frame variables.

`list_combine()` combines each element of a set of lists into a single list. If an element is not a list, it is treated as a length-one list. This happens to be very useful for combining data frame cols.

`list_drop_null()` removes `NULL` list elements very quickly.

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
```

---

math *Parallelised math operations*

Description

Parallelised math operations

Usage

```
abs_(x)

floor_(x)

ceiling_(x)

trunc_(x)
```

```
negate_(x)

exp_(x)

sqrt_(x)

sign_(x)

log_(x, base = exp(1))

log10_(x)

round_(x, digits = 0)

signif_(x, digits = 6)

add_(x, y)

subtract_(x, y)

multiply_(x, y)

divide_(x, y)

pow_(x, y)
```

Arguments

x	[numeric(n)] vector.
base	[numeric(n)] - Logarithm base.
digits	[numeric(n)] - Number of digits to round to.
y	[numeric(n)] vector.

Value

A transformed integer or double vector.

---

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.

`na_init` *Fast* `NA` *initialisation*

Description

Fast `NA` initialisation

Usage

```
na_init(x, n = 0L)
```

Arguments

`x`	A vector.
`n`	Vector length to initialise.

Value

Initialises `NA` values of the same type as `x`.

See Also

rep_len_

`new_df` *Cheap data frame utilities*

Description

Cheap data frame utilities

Usage

```
new_df(..., .nrows = NULL, .recycle = TRUE, .name_repair = TRUE, .args = NULL)

as_df(x)

fast_df(..., .args = NULL)

df_modify(x, cols)

list_as_df(x)

name_repair(x, dup_sep = "_", empty_sep = "col_")

unique_name_repair(x, dup_sep = "_", empty_sep = "col_")

col_c(..., .recycle = TRUE, .name_repair = TRUE, .args = NULL)

row_c(..., .args = NULL)
```

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 `TRUE`.
`.name_repair`	`[logical(1)]` - Should duplicate and empty names repaired and made unique? Default is `TRUE`.
`.args`	An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.
`x`	An object to coerce to a `data.frame` or a character vector for `unique_name_repair()`.
`cols`	A list of values to add or modify data frame `x`.
`dup_sep`	`[character(1)]` A separator to use between duplicate column names and their locations. Default is `'_'`
`empty_sep`	`[character(1)]` A separator to use between the empty column names and their locations. Default is `'col_'`

Details

`fast_df()` is a very fast bare-bones version of `new_df()` that performs no checks and no recycling or name tidying, making it appropriate for very tight loops.

Value

A `data.frame`.
`name_repair` takes a character vector and returns unique strings by appending duplicate string locations to the duplicates. This is mostly used to create unique col names.

new_logical *Fast multi-threaded vector initialisation*

Description

Fast multi-threaded vector initialisation

Usage

```
new_logical(n = 0L, names = NULL, default = FALSE)

new_integer(n = 0L, names = NULL, default = 0L)

new_double(n = 0L, names = NULL, default = 0)

new_character(n = 0L, names = NULL, default = "")

new_complex(n = 0L, names = NULL, default = complex(real = 0, imaginary = 0))

new_raw(n = 0L, names = NULL, default = as.raw(0))
```

Arguments

n	[integer(1)] - Length of vector.
names	[character(n)] - Names of initialised vector.
default	Default value to initialise the vector with.

Value

New vector.

See Also

new_list

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

Description

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

Usage

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

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

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

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

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

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

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

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

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

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

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

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

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

Arguments

x	A vector or data frame.
digits	How many decimal places should the summary statistics be printed as? Default is 2.
...	Further arguments passed onto methods. Currently unused.
hist	Should in-line histograms be returned? Default is FALSE.

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
```

`rebuild` *Rebuild an object from a template*

Description

Rebuild an object from a template

Usage

```
rebuild(x, template, ...)

## S3 method for class 'data.frame'
rebuild(x, template, shallow_copy = TRUE, ...)

## S3 method for class 'data.table'
rebuild(x, template, shallow_copy = TRUE, ...)

## S3 method for class 'tbl_df'
rebuild(x, template, shallow_copy = TRUE, ...)

## S3 method for class 'sf'
rebuild(x, template, shallow_copy = TRUE, ...)
```

Arguments

x	An object in which carefully selected attributes will be copied into from template.
template	A template object used to copy attributes into x.
...	Further arguments passed onto methods.
shallow_copy	Should x be shallow copied before rebuilding? Default is TRUE.

Details

In R attributes are difficult to work with. One big reason for this is that attributes may or may not be independent of the data. Date vectors for example have attributes completely independent of the data and hence if the attributes are removed at any point, they can easily be re-added without any calculations. Factors have almost data-independent attributes with an exception being when factors are combined. In some cases it is not possible to rebuild attributes from the data alone.

You can add your own rebuild method for an object not covered by the methods here.

Value

An object similar to template.

---

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, .args = NULL)
```

Arguments

...	Objects to recycle.
length	Optional length to recycle objects to.
.args	An alternative to ... so you can supply arguments directly in a list. This is equivalent to do.call(f, .args) but much more efficient.

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)

# Recycles both to size 10
recycle(Sys.Date(), 1:10)

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

# Unless length is supplied
recycle(integer(), 1:10, length = 10)

# Data frame rows are recycled
recycle(sset(iris, 1:3), length = 9)

# To recycle objects in a list, use `.args`
my_list <- list(from = 1L, to = 10L, by = seq(0.1, 1, 0.1))
recycle(.args = my_list)
```

---

rep *cheapr style repeat functions*

Description

cheapr style repeat functions

Usage

```
cheapr_rep(x, times)

rep_(x, times)

cheapr_rep_len(x, length)

rep_len_(x, length)

cheapr_rep_each(x, each)

rep_each_(x, each)
```

Arguments

x	A vector or data frame.
times	[integer(n)] A vector of times to repeat elements of x. Can be length 1 or the same length as vector_length(x).
length	[integer(1)] - Length of the recycled result.
each	[integer(n)] - How many times to repeat out each element of x.

Value

Repeated out object.

`replace` *Fast vector replacement, an alternative to* `[<-`

Description

Fast vector replacement, an alternative to `[<-`

Usage

```
replace_(x, where, with, in_place = FALSE, quiet = FALSE)
```

Arguments

`x`	A vector.
`where`	`[integer(n)]` - Where to assign replacement values. This can be an integer vector of locations, a logical vector (passed to `which_()`), or a character vector of names.
`with`	Replacement values. These will be recycled against the resulting `where` integer locations.
`in_place`	`[logical(1)]` - Should assignment be done in-place (no copies)? Default is `FALSE`. Please note that assignment will occur in-place where possible even if `in_place` is set to `FALSE`.
`quiet`	Should warnings be suppressed when `in_place = TRUE` and `x` is shared my multiple objects? Default is `FALSE`.

Value

A vector whose values are replaced with `with` at locations specified by `where`.

Examples

```
library(cheapr)

x <- set_round(seq_(-2, 2, by = 0.5))

x |>
  replace_(1, with = 100) # Assign value 100 at location 1

# Base R casts to `x` and replacement to a common type
`[<-`(x, x== 0, "42")

# `assign_at` only casts replacement to type of x
x |>
  replace_(x == 0, with = "42") # Assign value 42 where x == 0
```

sequences *Utilities for creating many sequences*

Description

seq_ is a vectorised version of seq with some additional features.
seq_size returns sequence sizes.
seq_start returns sequence start points.
seq_end returns sequence end points.
seq_increment returns sequence increments.
sequence_ is an extension to sequence which accepts decimal number increments.
seq_id can be paired with sequence_ to group individual sequences.
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, as_list = FALSE)

seq_id(size)

seq_(
  from = NULL,
  to = NULL,
  by = NULL,
  size = NULL,
  add_id = FALSE,
  as_list = FALSE
)

seq_size(from, to, by = 1L)

seq_start(size, to, by = 1L)

seq_end(size, from, by = 1L)

seq_increment(size, from, to)

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.
as_list	Should a list of sequences be returned? Setting to TRUE would place each distinct sequence vector into a distinct list element. The 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

seq_() is a fast vectorised version of seq() with powerful features. It can return many sequences as a single vector of combined sequences or a list of sequences.

sequence_() works in the same way as sequence() but can accept non-integer by values. This is the workhorse function of seq_().

Unlike sequence(), sequence_() recycles all its arguments, including size.

If any of the sequences contain values > .Machine$integer.max, then the result will always be a double vector.

Value

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

Examples

```
library(cheapr)

# These two functions are similar
sequence(1:3);sequence_(1:3)

# sequence_() can handle any numeric vector sequence
sequence(1:3, by = 0.1);sequence_(1:3, by = 0.1)

# Alternatively return as a list of sequences

sequence_(1:3, by = 0.1, as_list = TRUE)

# Add IDs to the sequences
sequence_(1:3, by = 0.1, add_id = TRUE)
# Turn this quickly into a data frame
seqs <- sequence_(1:3, by = 0.1, add_id = TRUE)
new_df(name = names(seqs), seq = seqs)
```

```
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 above
c(seq(1, 10, 1), seq(1, 10, 0.5))

# Again, as a list of sequences
# 2 different start points and 2 different increments
seq_(from = c(-1, 1), 3, by = c(1, 0.5), as_list = TRUE)

# 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 = " "))

# Or return as a list of sequences
# Note that these lengths will match the above line of code
seq_(1, 10, by = c(1, 0.5), as_list = TRUE) |>
  list_lengths()

# Sequences of dates with different increments

from <- Sys.Date()
to <- from + 10
by <- c(1, 2, 3)
date_seqs <- seq_(from, to, by, as_list = TRUE)
lapply(date_seqs, function(x) `class<-`(x, "Date"))

# Utilities for rolling calculations

# A window sequence of size 3 for a vector of size 10
# This tells us how big the window should be when looking backwards
window_sequence(10, 3, partial = FALSE)
window_sequence(10, 3, partial = TRUE)

window_sequence(c(3, 5), 3)
window_sequence(c(3, 5), 3, partial = FALSE)
window_sequence(c(3, 5), 3, partial = TRUE, ascending = FALSE)

# Lag sequence of size 3 for a vector of size 10
# This tells us how for we should look backwards at any given point
lag_sequence(10, 3, partial = FALSE)
# How far to look forwards
lead_sequence(10, 3, partial = FALSE)

lag_sequence(10, 3, partial = TRUE)
lead_sequence(10, 3, partial = TRUE)
# 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)

x %in_% table

x %!in_% table

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)

cheapr_var(x, na.rm = TRUE)

cheapr_rev(x)

cheapr_sd(x, na.rm = TRUE)

rev_(x)

sd_(x, na.rm = TRUE)

var_(x, na.rm = TRUE)

with_local_seed(expr, .seed = NULL, .envir = environment(), ...)
```

Arguments

x	A vector or data frame.
y	A vector or data frame.
dups	Should duplicates be kept? Default is TRUE.
table	See ?collapse::fmatch
size	See ?sample.

replace	See ?sample.
prob	See ?sample.
value	The column name to assign the values of a vector.
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.
na.rm	Should NA values be ignored in var_() Default is TRUE.
expr	Expression that will be evaluated with a local seed that is independent and has absolutely no effect on the global RNG state.
.seed	A local seed to set which is only used inside with_local_seed(). After the execution of the expression the original seed is reset.
.envir	Environment to evaluate expression.
...	Further arguments passed onto cut or set.seed.

Value

intersect_() returns a vector of common values between x and y.
setdiff_() returns a vector of values in x but not y.
%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.
var_ returns the variance of a numeric vector. No coercion happens for integer vectors and so is very cheap.
rev_ is a much cheaper version of rev().
with_local_seed offers no speed improvements but is extremely handy in executing random number based expressions like rnorm() without affecting the global RNG state. It allows you to run these expressions in a sort of independent 'container' and with an optional seed for that 'container' for reproducibility. The rationale for including this in 'cheapr' is that it can reduce the need to set many seed values, especially for multiple output comparisons of RNG expressions. Another way of thinking about it is that with_local_seed() is a helper that allows you to write reproducible code without side-effects, which traditionally cannot be avoided when calling set.seed() directly.

Examples

```
library(cheapr)

# Using `with_local_seed()`

# The below 2 statements are equivalent

# Statement 1
set.seed(123456789)
res <- rnorm(10)
```

```
# Statement 2
res2 <- with_local_seed(rnorm(10), .seed = 123456789)

# They are the same
identical(res, res2)

# As an example we can see that the RNG is unaffected by generating
# random uniform deviates in batches between calls to `with_local_seed()`
# and comparing to the first result

set.seed(123456789)
batch1 <- rnorm(2)

with_local_seed(runif(10))
batch2 <- rnorm(2)
with_local_seed(runif(10))
batch3 <- rnorm(1)
with_local_seed(runif(10))
batch4 <- rnorm(5)

# Combining the batches produces the same result
# therefore `with_local_seed` did not interrupt the rng sequence
identical(c(batch1, batch2, batch3, batch4), res)

# It can be useful in multiple comparisons
out1 <- with_local_seed(rnorm(5))
out2 <- with_local_seed(rnorm(5))
out3 <- with_local_seed(rnorm(5))

identical(out1, out2)
identical(out1, out3)
```

`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	[numeric(n)] vector.
digits	[numeric(n)] - Number of digits to round to.
base	[numeric(n)] - Logarithm base.
y	[numeric(n)] 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)
mark(
  base = exp(log(abs(x))),
  cheapr = set_exp(set_log(set_abs(x)))
)
```

---

sset *Cheaper subset* `sset()`

Description

`sset()` is a cheaper alternative to `[`.

It consistently subsets data frame rows for any data frame class including tibble and data.table.

Usage

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

Arguments

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

Details

S3 dispatching:

`sset` will internally dispatch the correct method and will call `[` if it can't find an appropriate method. This means one can define their own `[` method for custom S3 objects.

To speed up subsetting for common objects likes Dates and `POSIXlt` an internal generic function is used which overwrites the `[` method for that common object. This is why subsetting `POSIXlt` is much faster with `sset` an internal method has been defined. For more details see the code for `cheapr:::cheapr_sset`.

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
```

`sset_df` *Fast functions for data frame subsetting*

Description

These functions are for developers that need minimal overhead when filtering on rows and/or cols.

Usage

```
sset_df(x, i = NULL, j = NULL, ...)

sset_row(x, i = NULL)

sset_col(x, j = NULL)
```

Arguments

x	A data.frame.
i	Rows - If NULL all rows are returned.
j	Cols - If NULL all cols are returned.
...	Unused.

Details

If you are unsure which functions to use then it is recommended to use sset(). These low-overhead helpers do not work well with data.tables but should work well with basic data frames and basic tibbles. The only real difference between sset_df and sset_row/sset_col is that sset_df attempts to return a similar type of data frame as the input, whereas sset_row and sset_col always return a plain data frame.

Value

A data frame subsetted on rows i and cols j.

---

strings *Fast string concatenation using C++*

Description

Fast string concatenation using C++

Usage

```
paste_(..., sep = "", collapse = NULL, .args = NULL)
```

Arguments

...	Character vectors to concatenate.
sep	[character(1)] - A string to separate the supplied strings.
collapse	Optional string to collapse concatenated strings into one string (character vector of length 1).
.args	An alternative to ... so you can supply arguments directly in a list. This is equivalent to do.call(f, .args) but much more efficient.

Examples

```
library(cheapr)
# Normal usage
paste_("Hello", "and", "Goodbye", sep = " ")
paste_(100, "%")

paste_(letters, LETTERS)
paste_(letters, LETTERS, collapse = "")

# Both concatenating and collapsing
paste_(letters, LETTERS, sep = ",", collapse = " next letter ")
# This is the same as above
paste_(letters, LETTERS, collapse = ", next letter ")

# Recycling with zero-length vectors
paste_("hello", character(), letters)
paste_("hello", character(), letters, collapse = "")

library(bench)

sampled_letters <- sample_(letters, 5e04, TRUE)

# Pasting multiple character vectors
mark(
  paste_(sampled_letters, sep = ","),
  paste(sampled_letters, sep = ","),
  iterations = 50
)
# Collapsing is very fast compared to base R
mark(
  paste_(sampled_letters, collapse = ""),
  paste(sampled_letters, collapse = ""),
  iterations = 50
)

# Concatenating many objects is very fast via `.args`

strings <- sampled_letters |>
  with_local_seed(1) |>
  as.list()

strings <- lapply(strings, rep_len_, 3)

mark(
  paste_(.args = strings),
  do.call(paste0, strings),
  iterations = 15
)
mark(
  paste_(.args = strings, collapse = ""),
  do.call(paste_, c(strings, list(collapse = ""))),
  do.call(paste0, c(strings, list(collapse = ""))),
```

```
  iterations = 10
)
```

`str_coalesce` *Coalesce character vectors*

Description

`str_coalesce()` find the first non empty string `""`. This is particularly useful for assigning and fixing the names of R objects.

In this implementation, the empty string `""` has priority over `NA` which means `NA` is only returned when all values are `NA`, e.g. `str_coalesce(NA, NA)`.

Usage

```
str_coalesce(..., .args = NULL)
```

Arguments

`...`	Character vectors to coalesce.
`.args`	An alternative to `...` so you can supply arguments directly in a list. This is equivalent to `do.call(f, .args)` but much more efficient.

Details

`str_coalesce(x, y)` is equivalent to `if_else(x != "" & !is.na(x), x, y)`.

Value

A coalesced character vector of length corresponding to the recycled size of supplied character vectors. See `?recycle` for details.

Examples

```
library(cheapr)

# Normal examples
str_coalesce("", "hello")
str_coalesce("", NA, "goodbye")

# '' always preferred
str_coalesce("", NA)
str_coalesce(NA, "")

# Unless there are only NAs
str_coalesce(NA, NA)

# `str_coalesce` is vectorised
```

```
x <- val_insert(letters, "", n = 10)
y <- val_insert(LETTERS, "", n = 10)

str_coalesce(x, y)

# Using `.args` instead of `do.call` is much more efficient
library(bench)
x <- rep_len_(list(letters), 10^3)

mark(do.call(str_coalesce, x),
     str_coalesce(.args = x),
     iterations = 50)
```

`switch_args` *Switch between dot-dot-dot and a list of args*

Description

`switch_args()` is primarily used as a helper when writing functions that use the dot-dot-dot argument `...`.
cheapr uses it frequently to give more freedom to the user in how they pass arguments to functions that take a variable number of arguments.
See examples for info.

Usage

```
switch_args(..., .args = NULL)
```

Arguments

`...`	Arguments passed individually.
`.args`	Alternative list of arguments. Either `...` or `.args` can be used, not both.

Details

Using `switch_args` simply allows the user to avoid having to use `do.call(fn, args)`. This can be advantageous for developers who write compiled (C/C++) functions that accept lists of arguments. cheapr internally uses this framework for performance critical functions such as `cheapr::c_()` which internally calls `cheapr:::cpp_c()`, a function that takes one list of vectors and combines them into one vector. The equivalent of `cheapr::c_(.args = args)` would be the less efficient `do.call(cheapr::c_, args)`.

Value

A list of arguments

Examples

```
library(cheapr)

# Flexibly create a data frame
base_new_df <- function(..., .args = NULL){

  args <- switch_args(..., .args = .args)

  list2DF(args)
}

# Normal usage
base_new_df(x = 1, y = 2)

# Alternatively supplying a list of args instead
base_new_df(.args = list(x = 1, y = 2))

# cheapr::new_df does something similar
new_df(x = 1, y = 2)
new_df(.args = list(x = 1, y = 2))
```

---

unique_ *An alternative* unique *function*

Description

`unique_()` is a usually faster alternative to `unique()` with optional sorting included. The internal API of this function is designed to be simple and generic to allow for working with all kinds of objects that can be reduced to a unique set.

Internally `unique_()` calculates unique group IDs for the given vector in the range `[1, n]` where `1` denotes the first group and `n` denotes the nth group. This function will work correctly as long as there is a correctly implemented `collapse::GRP` method and a `[` method for the object. In the future cheapr will include a `group_id` S3 generic to replace the use of `collapse::GRP` here, of which is arguably more difficult to write correct methods for.

Usage

```
unique_(x, sort = FALSE)
```

Arguments

`x`	A vector (or data frame).
`sort`	Should unique result be sorted? Default is `FALSE`.

Value

A unique vector (or data frame).

Examples

```
library(cheapr)

x <- rep_(3:1, 3)
unique_(x)
unique_(x, sort = TRUE)

# Unique rows
iris |>
  sset(j = c("Petal.Width", "Species")) |>
  unique_()
```

`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)

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