Package 'rray'

July 23, 2019

Title Simple Arrays

Version 0.1.0

Description Provides a toolkit for manipulating arrays in a consistent, powerful, and intuitive manner through the use of broadcasting and a new array class, the 'rray'.

License GPL-3

URL https://github.com/r-lib/rray

BugReports https://github.com/r-lib/rray/issues

Depends R (>= 3.2)

Imports glue, Rcpp (>= 1.0.1), rlang (>= 0.4.0), utils, vctrs (>= 0.2.0)

Suggests covr, knitr, magrittr, rmarkdown, testthat (>= 2.1.0)

LinkingTo Rcpp, xtensor (>= 0.11.1-0)

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

NeedsCompilation yes

Author Davis Vaughan [aut, cre], RStudio [cph]

Maintainer Davis Vaughan <davis@rstudio.com>

Repository CRAN

Date/Publication 2019-07-23 12:10:03 UTC

R topics documented:

as_array *Coerce to an array*

Description

`as_array()` coerces `x` to an array. `x` will keep any existing dimensions and dimension names.

Usage

```
as_array(x, ...)
```

Arguments

`x`	An object to coerce to an array.
`...`	Objects passed on to methods.

Value

An array.

See Also

`as_matrix()`

Examples

```
as_array(1:10)
```

as_matrix *Coerce to a matrix*

Description

`as_matrix()` coerces `x` to a matrix.

Usage

```
as_matrix(x, ...)
```

Arguments

`x`	An object to coerce to a matrix.
`...`	Objects passed on to methods.

Details

1D arrays are coerced to 1 column matrices.

For a >2D object to be coercible to a matrix, all of the dimensions except for the first two must be size 1. Meaning an array with dimensions `(3,2,1)` would be coercible to a `(3,2)` matrix, but one with `(3,1,2)` would not be.

Value

A matrix.

See Also

`as_array()`

Examples

```
as_matrix(rray(1:10))

# >2D structures can be coerced to matrices
# their first and second dimensions are
# the only ones having a size >1
x <- rray(1, c(2, 2, 1))
as_matrix(x)

# This cannot be coerced to a matrix
y <- rray_reshape(x, c(2, 1, 2))
try(as_matrix(y))
```

---

as_rray *Coerce to an rray*

Description

Coerce `x` to an rray. It will keep its dimensions and dimension names if it has any.

Usage

```
as_rray(x)
```

Arguments

`x`	An object to coerce to an rray.

Value

An rray.

Examples

```
as_rray(1)

ex <- matrix(1:10, nrow = 5, dimnames = list(NULL, c("a", "b")))
as_rray(ex)
```

`common-dim-names` *Find common dimension names*

Description

Obtain a list of common dimension names among a set of objects. For interactive use, `rray_dim_names_common()` is more useful.

Usage

```
rray_dim_names_common(...)

rray_dim_names2(x, y)
```

Arguments

`x, y, ...`	Objects to find common dimensions for.

Details

`rray_dim_names_common()` is the engine that determines what dim names should be used in the result of arithmetic operations and other functions that involve multiple rray objects and return a single result.

The rules for determining the set of common dim names between objects `x` and `y` (in that order) are:

- Compute the common `dim` between `x` and `y` using `rray_dim_common()`.
- For each axis along the common `dim`:
  - If `x` has names for that axis *and* the size of the names vector is the same as the size of the axis, use those names for that axis.
  - Else if `y` has names for that axis *and* the size of the names vector is the same as the size of the axis, use those names for that axis.
  - Otherwise, the names for that axis is `NULL`.

Value

A list of the common dimension names of the inputs.

Examples

```
library(magrittr)

# 1x2 - Row names but no column names
x <- rray(1, dim = c(1, 2)) %>%
  rray_set_row_names("r_from_x")

# 1x1 - Row names and column names
y <- rray(1, dim = c(1, 1)) %>%
  rray_set_col_names("c_from_y") %>%
  rray_set_row_names("r_from_y")

# 1x1 - Row names but no column names
z <- rray(1, c(1, 1)) %>%
  rray_set_row_names("r_from_z")

# Combining y and z
# y has names for both dimensions
# so they are used
rray_dim_names_common(y, z)

# Combining z and y
# - Row names are found first from z
# - But z has no column names
# - So column names are found from y
rray_dim_names_common(z, y)

# Combining x and y
# - Row names are found first from x
# - x has no column names
# - y has column names but they are
#   a different length from the common
#   column dimension (common size of 2)
# - So no column names are used
rray_dim_names_common(x, y)
```

---

dim-names *Dimension names*

Description

This family of functions allows you to get and set dimension names in various ways.

- `rray_dim_names()` returns a list of the dimension names.
- `rray_axis_names()` returns a character vector or `NULL` containing the names corresponding to the `axis` dimension.
- `rray_row_names()` and `rray_col_names()` are helpers for getting the row and column names respectively.

- Each of these four functions also has "set" variants: a functional form (i.e. `rray_set_row_names()`), and an assignment form (i.e. `rray_row_names<-()`).

Usage

```
rray_dim_names(x)

rray_dim_names(x) <- value

rray_set_dim_names(x, dim_names)

rray_axis_names(x, axis)

rray_axis_names(x, axis) <- value

rray_set_axis_names(x, axis, names, meta = NULL)

rray_row_names(x)

rray_row_names(x) <- value

rray_set_row_names(x, names, meta = NULL)

rray_col_names(x)

rray_col_names(x) <- value

rray_set_col_names(x, names, meta = NULL)
```

Arguments

`x`	The object to extract the dimension names for.
`value`	For `rray_dim_names<-()`, a list containing either character vectors or `NULL` corresponding to the new dimension names to use for `x`. Otherwise, identical to `names`.
`dim_names`	A list of either character vectors or `NULL` representing the new dim names of `x`. If `NULL` is supplied, the dim names of `x` are removed.
`axis`	A single integer. The axis to select dimension names for.
`names`	A character vector of new dimension names for the `axis` dimension. This is also allowed to be `NULL` to remove dimension names for the specified axis.
`meta`	A single character representing an optional "meta" name assigned to the `axis` names. If `NULL`, the current meta name is kept.

Details

Unlike `dimnames()` which can return `NULL`, `rray_dim_names()` always returns a list the same length as the dimensionality of `x`. If any dimensions do not have names, `NULL` is returned for

that element of the list. This results in an object that's length always matches the dimensionality of `x`.

Value

`rray_dim_names()` returns a list of dimension names. The other names functions return character vectors, or `NULL`, corresponding to the names of a particular axis.

Examples

```
x <- rray(1:10, c(5, 2))
rray_dim_names(x) <- list(letters[1:5], NULL)
x
rray_dim_names(x)

# 3D object, so 3 dim name elements
rray_dim_names(rray(1, dim = c(1, 1, 1)))

# Vectors are treated as 1D arrays
vec <- c(x = 1, y = 2)
rray_dim_names(vec)

# You can add dim names more easily
# using rray_set_axis_names()
# and the pipe operator
library(magrittr)
y <- rray(1, c(1, 2, 1)) %>%
  rray_set_axis_names(1, "r1") %>%
  rray_set_axis_names(2, c("c1", "c2")) %>%
  rray_set_axis_names(3, "3rd dim")

y

# You can set also set axis names to `NULL` to reset them
rray_set_axis_names(y, 2, NULL)

# You can set the "meta" names of an axis as well
rray_set_axis_names(y, 1, "r1", "row names")
```

`extremum` *Maximum and minimum values*

Description

`rray_maximum()` and `rray_minimum()` compute the elementwise max / min between `x` and `y`.

Usage

```
rray_maximum(x, y)

rray_minimum(x, y)
```

Arguments

x, y	A vector, matrix, array or rray.

Value

The elementwise max/min of x and y, with broadcasting.

Examples

```
# Elementwise maximum
rray_maximum(c(1, 2, 3), c(3, 2, 1))

# Elementwise minimum
rray_minimum(c(1, 2, 3), c(3, 2, 1))

# With broadcasting
x <- matrix(1:3)
rray_maximum(x, t(x))
```

---

is_rray *Is* x *an rray?*

Description

is_rray() tests if x is an rray object.

Usage

```
is_rray(x)
```

Arguments

x	An object.

Value

A single logical.

Examples

```
is_rray(rray(1:5))

is_rray(1:5)
```

`new_rray` *Create a new rray*

Description

Low level constructor for rray objects

Usage

```
new_rray(.data = numeric(0), size = 0L, shape = integer(0),
  dim_names = NULL, ..., subclass = character(0))
```

Arguments

`.data`	A numeric vector with no attributes representing the data.
`size`	An integer. The number of *observations* in the object. This is equivalent to the number of rows.
`shape`	An integer vector. The shape corresponds to all of the dimensions in the object except for the first one (the `size`).
`dim_names`	A list. For no names, `NULL`, in which case a list of empty characters will be constructed. Otherwise the list must be the same length as the total number of dimensions (i.e `length(c(size,shape))`). Each element of the list much be either a character vector the same size as the corresponding dimension, or `character(0)` for no names for that dimension.
`...`	Name-value pairs defining attributes.
`subclass`	The name of the subclass.

Value

A new rray.

Examples

```
rray_ex <- new_rray(
  .data = 1:10,
  size = 5L,
  shape = 2L,
  dim_names = list(character(), c("a", "b"))
```

```
)

rray_ex
```

`pad` *Pad missing dimensions when subsetting*

Description

`pad()` is used alongside the standard rray subsetting operator `[` (and the underlying `rray_subset()` function) to easily subset into higher dimensions without having to explicitly list the intermediate commas.

Usage

```
pad()
```

Value

An object that can be used to pad dimensions with when subsetting.

Examples

```
x <- rray(1:4, c(1, 1, 2, 2))

# pad() fills in the missing dimensions
# essentially it adds commas automatically

# second element in the 4th dimension
x[pad(), 2]

# vs using
x[,,,2]

# second element in 3rd
# first element in 4th
x[pad(), 2, 1]

# can fill in the missing gaps too
# this fills in the 2nd/3rd dimensions
x[1, pad(), 1]

# if a pad() isn't needed
# because the dimensionality is already fully
# specified by the indices, its ignored
x_flat <- rray_reshape(x, 4)
x_flat[pad(), 1]
x_flat[1, pad()]
```

```
# `pad()` can be used with base R
# objects as well through `rray_subset()`
x_arr <- as.array(x)
rray_subset(x_arr, pad(), 1)
```

`rray` *Build a rray object*

Description

Constructor for building rray objects. Existing vectors, matrices, and arrays can be used to build the rray, but their dimension names are not retained.

Usage

```
rray(x = numeric(0), dim = NULL, dim_names = NULL)
```

Arguments

`x`	A numeric vector, matrix, or array to convert to an rray.
`dim`	An integer vector describing the dimensions of the rray. If `NULL`, the dimensions are taken from the existing object using `rray_dim()`.
`dim_names`	A list. For no names, `NULL`, in which case a list of empty characters will be constructed. Otherwise the list must be the same length as the total number of dimensions (i.e `length(c(size,shape))`). Each element of the list much be either a character vector the same size as the corresponding dimension, or `character(0)` for no names for that dimension.

Details

The `dim` argument is very flexible.

- If `vec_size(x) == prod(dim)`, then a reshape is performed.
- Otherwise broadcasting is attempted.

This allows quick construction of a wide variety of structures. See the example section for more.

rray objects are never reduced to vectors when subsetting using `[` (i.e. dimensions are never dropped).

Value

An rray.

Examples

```
# 1D rray. Looks like a vector
# functions similar to a 1 column matrix
rray(c(1,2,3), dim = c(3))

# 3 rows, 4 cols
rray(c(1,2,3), dim = c(3, 4))

# 3x2x4 array
rray(1, dim = c(3, 2, 4))

# from a matrix
mat <- matrix(c(1, 2, 3, 4), ncol = 2)
rray(mat)

# from a matrix, with broadcasting
rray(mat, dim = c(2, 2, 3))

# reshape that matrix during creation
# (different from broadcasting)
rray(mat, dim = c(1, 4))

# from an array, with broadcasting
arr <- array(1, c(1, 2, 2))
rray(arr, c(3, 2, 2))

# with row names
rray(c(1, 2, 3), c(3, 2), dim_names = list(c("x", "y", "z"), NULL))
```

`rray-compare` *Compare arrays*

Description

These operators compare multiple arrays together, with broadcasting. The underlying functions powering the comparison operators are also exported for use with base R objects.

Usage

```
## S3 method for class 'vctrs_rray'
e1 > e2

rray_greater(x, y)

## S3 method for class 'vctrs_rray'
e1 >= e2
```

```
rray_greater_equal(x, y)

## S3 method for class 'vctrs_rray'
e1 < e2

rray_lesser(x, y)

## S3 method for class 'vctrs_rray'
e1 <= e2

rray_lesser_equal(x, y)

## S3 method for class 'vctrs_rray'
e1 == e2

rray_equal(x, y)

## S3 method for class 'vctrs_rray'
e1 != e2

rray_not_equal(x, y)
```

Arguments

`e1, e2`	Generally, the same as `x` and `y`. See Details.
`x, y`	Two vectors, matrices, arrays, or rrays.

Details

The comparison operators themselves rely on R's dispatching rules to dispatch to the correct rray comparison operator. When comparing rrays with base R matrices and arrays, this generally works fine. However, if you compare classed objects like `factor("x") > rray(1)` then a fall through method is used and a warning is thrown. There is nothing we can do about this. See `?groupGeneric` for more information on this.

Value

The value of the comparison, with dimensions identical to the common dimensions of the inputs.

Examples

```
x <- rray(1:12, c(2, 2, 3))
y <- matrix(1:2)

# True except in first 2 positions
x > y

# All true
x >= y
```

```
# False everywhere
x < y

# False except in the first 2 positions
x <= y
```

`rray-logical` *Logical operators*

Description

These functions perform logical operations on arrays, with broadcasting. They power the logical operators of `&`, `|`, and `!` with rrays, but are also exported for use with base R objects.

Usage

```
rray_logical_and(x, y)

rray_logical_or(x, y)

rray_logical_not(x)

rray_any(x, axes = NULL)

rray_all(x, axes = NULL)
```

Arguments

`x, y`	Vectors, matrices, arrays, or rrays.
`axes`	An integer vector specifying the axes to reduce over. `1` reduces the number of rows to `1`, performing the reduction along the way. `2` does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Details

The operators themselves rely on R's dispatching rules to dispatch to the correct rray logical operator. When comparing rrays with base R matrices and arrays, this generally works fine. However, if you compare classed objects like `factor("x") & rray(1)` then a fall through error is thrown. There is nothing we can do about this. See `?groupGeneric` for more information on this.

The behavior of comparing an array with a length 0 dimension with another array is slightly different than base R since broadcasting behavior is well defined. Length 0 dimensions are not exceptions to the normal broadcasting rules. Comparing dimensions of `0` and `1`, the common dimension is `0` because `1` always becomes the other dimension in the comparison. On the other hand, comparing dimensions `0` and `2` is an error because neither are `1`, and they are not identical.

Value

The value of the logical comparison, with broadcasting.

`rray_any()` and `rray_all()` return a logical object with the same shape as `x` everywhere except along `axes`, which have been reduced to size 1.

Examples

```
x <- rray(TRUE, c(2, 2, 3))
y <- matrix(c(TRUE, FALSE))

# `TRUE` wherever `y` is broadcasted to be `TRUE`
x & y

# ------------------------------------------------------------------------
# Behavior with edge cases

x <- rray(TRUE, c(1, 2))

# The common dim is (0, 2)
logical() & x

# You can have empty arrays with shape
# The common dim is (0, 2, 2)
y <- array(logical(), c(0, 1, 2))
x & y

# You cannot broadcast dimensions
# of 2 and 0. Following standard
# broadcasting rules, they do not
# match and neither are 1, so an
# error should be thrown
try(x & array(logical(), c(1, 0)))
```

---

rray_all_equal *Strictly compare arrays*

Description

Unlike `rray_equal()` and `rray_not_equal()`, these functions perform a strict comparison of two arrays, and return a single logical value. Specifically:

- Broadcasting is *not* performed here, as the shape is part of the comparison.
- The underlying type of the values matter, and `1` is treated as different from `1L`.
- Otherwise, attributes are not compared, so dimension names are ignored.

Usage

```
rray_all_equal(x, y)

rray_any_not_equal(x, y)
```

Arguments

x, y	Vectors, matrices, arrays, or rrays.

Examples

```
# This is definitely true!
rray_all_equal(1, 1)

# Different types!
rray_all_equal(1, 1L)

# Different types!
rray_all_equal(rray(1), matrix(1))

# Different shapes!
rray_all_equal(matrix(1), matrix(1, nrow = 2))

# Are any values different?
rray_any_not_equal(c(1, 1), c(1, 2))

# Is the shape different?
rray_any_not_equal(1, c(1, 2))

# Dimension names don't matter
x <- matrix(1, dimnames = list("foo", "bar"))
rray_all_equal(x, matrix(1))
```

---

rray_arith *Arithmetic operations*

Description

These functions provide the implementations for their underlying infix operators (i.e. `rray_add()` powers +). All operators apply broadcasting to their input.

Usage

```
x %b+% y

rray_add(x, y)

x %b-% y
```

```
rray_subtract(x, y)

x %b*% y

rray_multiply(x, y)

x %b/% y

rray_divide(x, y)

x %b^% y

rray_pow(x, y)

rray_identity(x)

rray_opposite(x)
```

Arguments

x, y	A pair of vectors.

Details

In case you want to apply arithmetic operations with broadcasting to purely base R objects using infix operators, custom infix functions have been exported, such as `%b+%`, which will perform addition with broadcasting no matter what type the input is.

Value

The value of the arithmetic operation, with dimensions identical to the common dimensions of the input.

Examples

```
library(magrittr)

x <- rray(1:8, c(2, 2, 2)) %>%
  rray_set_row_names(c("r1", "r2")) %>%
  rray_set_col_names(c("c1", "c2"))

y <- matrix(1:2, nrow = 1)

# All arithmetic functions are applied with broadcasting
rray_add(x, y)

# And the power `+` when any underlying input
# is an rray
x + y
```

```
# If you happen to only have base R matrices/arrays
# you can use `rray_add()` or `%b+%` to get the
# broadcasting behavior
rray_add(y, matrix(1:2))

y %b+% matrix(1:2)
```

`rray_bind` *Combine many arrays together into one array*

Description

These functions bind multiple vectors, matrices, arrays, or rrays together into one, combining along the `.axis`.

Usage

```
rray_bind(..., .axis)

rray_rbind(...)

rray_cbind(...)
```

Arguments

`...`	Vectors, matrices, arrays, or rrays.
`.axis`	A single integer. The axis to bind along.

Details

`rray_bind()` is extremely flexible. It uses broadcasting to combine arrays together in a way that the native functions of `cbind()` and `rbind()` cannot. See the examples section for more explanation!

Value

An array, or rray, depending on the input.

Examples

```
# ------------------------------------------------------------------------
a <- matrix(1:4, ncol = 2)
b <- matrix(5:6, ncol = 1)

# Bind along columns
rray_bind(a, b, .axis = 2)

# Bind along rows
```

```
# Broadcasting is done automatically
rray_bind(a, b, .axis = 1)

# Notice that this is not possible with rbind()!
try(rbind(a, b))

# You can bind "up" to a new dimension
# to stack matrices into an array
rray_bind(a, b, .axis = 3)

# ------------------------------------------------------------------------
# Dimension name example

x <- matrix(
 1:6,
 ncol = 3,
 dimnames = list(c("a_r1", "a_r2"), c("a_c1", "a_c2", "a_c3"))
)

y <- matrix(
 7:8,
 ncol = 1,
 dimnames = list(NULL, c("b_c1"))
)

# Dimension names come along for the ride
# following rray name handling
rray_bind(x, y, .axis = 2)

# If some inputs are named, and others
# are not, the non-named inputs get `""`
# as names
rray_bind(x, y, .axis = 1)

# You can add "outer" names to the
# axis you are binding along.
# They are added to existing names with `..`
rray_bind(outer = x, y, .axis = 2)

# Outer names can be used to give unnamed
# inputs default names
rray_bind(outer = x, outer_y = y, .axis = 1)
```

`rray_broadcast` *Broadcast to a new dimension*

Description

`rray_broadcast()` will *broadcast* the dimensions of the current object to a new dimension.

Usage

```
rray_broadcast(x, dim)
```

Arguments

x	The object to broadcast.
dim	An integer vector. The dimension to broadcast to.

Details

Broadcasting works by *recycling dimensions* and *repeating values* in those dimensions to match the new dimension.

Here's an example. Assume you have a 1x3 matrix that you want to broadcast to a dimension of 2x3. Since the 1st dimensions are different, and one of them is 1, the 1 row of the 1x3 matrix is repeated to become a 2x3 matrix. For the second dimension, both are already 3 so nothing is done.

```
(1, 3)
(2, 3)
------
(2, 3)
```

As an example that *doesn't* broadcast, here is an attempt to make a 2x1x4 matrix broadcast to a 2x3x5 matrix (In the R world, 2x3x4 means a 2 row, 3 column, and 4 "deep" array). The first 2 dimensions are fine, but for the third dimension, 4 and 5 are not "recyclable" and are therefore incompatible.

```
(2, 1, 4)
(2, 3, 5)
---------
(2, 3, X)
```

You can broadcast to higher dimensions too. If you go from a 5x2 to a 5x2x3 array, then the 5x2 matrix implicitly gets a 1 appended as another dimension (i.e. 5x2x1)

```
(5, 2,  ) <- implicit 1 is recycled
(5, 2, 3)
---------
(5, 2, 3)
```

Broadcasting is an important concept in rray, as it is the engine behind the different structure for arithmetic operations.

Value

x broadcast to the new dimensions.

Examples

```
# From 5x1 ...
x <- rray(1:5)

# ...to 5x2
rray_broadcast(x, c(5, 2))

# Internally, rray() uses broadcasting
# for convenience so you could have also
# done this with:
rray(1:5, dim = c(5, 2))

# Moar dimensions
rray_broadcast(x, c(5, 2, 3))

# You cannot broadcast down in dimensions
try(rray_broadcast(x, 5))
```

---

rray_clip *Bound the values of an array*

Description

`rray_clip()` sets *inclusive* lower and upper bounds on the values of `x`.

Usage

```
rray_clip(x, low, high)
```

Arguments

`x`	A vector, matrix, array or rray.
`low`	A single value. The lower bound. `low` is cast to the inner type of `x`.
`high`	A single value. The upper bound. `high` is cast to the inner type of `x`.

Value

`x` bounded by `low` and `high`.

Examples

```
# bound `x` between 1 and 5
x <- matrix(1:10, ncol = 2)
rray_clip(x, 1, 5)
```

`rray_diag` *Create a matrix with* `x` *on the diagonal*

Description

`rray_diag()` creates a matrix filled with `x` on the diagonal. Use `offset` to place `x` along an offset from the diagonal.

Usage

```
rray_diag(x, offset = 0)
```

Arguments

`x`	A vector, matrix, array or rray.
`offset`	A single integer specifying the offset from the diagonal to place `x`. This can be positive or negative.

Details

No dimension names will be on the result.

Value

A matrix, with `x` on the diagonal.

Examples

```
# Creates a diagonal matrix
rray_diag(1:5)

# Offset `1:5` by 1
rray_diag(1:5, 1)

# You can also go the other way
rray_diag(1:5, -1)

# Identity matrix
rray_diag(rep(1, 5))

# One interesting use case of this is to create
# a square empty matrix with dimensions (offset, offset)
rray_diag(rray(integer()), 3)
rray_diag(logical(), 3)
```

`rray_dim` *Find common dimensions*

Description

- `rray_dim()` finds the dimension of a single object.
- `rray_dim_common()` finds the common dimensions of a set of objects.

Usage

```
rray_dim(x)

rray_dim_common(...)
```

Arguments

`x`	An object.
`...`	Objects to find common dimensions for.

Details

`rray_dim_common()` first finds the common dimensionality, makes any implicit dimensions explicit, then recycles the size of each dimension to a common size.

As an example, the common dimensions of `(4,5)` and `(4,1,2)` are:

```
(4, 5, 1) <- implicit 1 is made to be explicit, then recycled to 2
(4, 1, 2) <- the 1 in the second dimension here is recycled to 5
---------
(4, 5, 2) <- resulting common dim
```

The resulting dimensions from `rray_dim_common()` are the dimensions that are used in broadcasted operations.

Value

An integer vector containing the common dimensions.

See Also

`rray_dim_n()`

Examples

```
x_1_by_4 <- rray(c(1, 2, 3, 4), c(1, 4))
x_5_by_1 <- rray(1:5, c(5, 1))

rray_dim(x_1_by_4)

# recycle rows: 1 VS 5 = 5
# recycle cols: 4 VS 1 = 4
rray_dim_common(x_1_by_4, x_5_by_1)

x_5_by_1_by_3 <- rray(1, c(5, 1, 3))

# recycle rows:    5 VS 1 = 5
# recycle cols:    4 VS 1 = 4
# recycle 3rd dim: 1 VS 3 = 3
# (here, 3rd dim of 1 for the matrix is implicit)
rray_dim_common(x_1_by_4, x_5_by_1_by_3)

# The dimensions of NULL are 0
rray_dim(NULL)
```

---

rray_dim_n *Compute the number of dimensions of an object*

Description

`rray_dim_n()` computes the dimensionality (i.e. the number of dimensions).

Usage

```
rray_dim_n(x)
```

Arguments

`x`	An object.

Details

One point worth mentioning is that `rray_dim_n()` is very strict. It does not simply call the generic function `dim()` and then check the length. Rather, it explicitly pulls the attribute for the `"dim"`, and checks the length of that. If an object does not have an attribute, then the dimensionality is 1.

This means that data frames have a dimensionality of 1, even though `dim()` defines a method for data frames that would imply a dimensionality of 2.

Value

An integer vector containing the number of dimensions of `x`.

Examples

```
x_1_by_4 <- rray(c(1, 2, 3, 4), c(1, 4))

rray_dim_n(x_1_by_4)

# NULL has a dimensionality of 1
rray_dim_n(NULL)

# The dimensionality of a data frame is 1
rray_dim_n(data.frame())
```

---

rray_dot *Matrix multiplication*

Description

`rray_dot()` works exactly like the base R function, `%*%`, but preserves the rray class where applicable. For the exact details of how 1D objects are promoted to 2D objects, see %*%.

Usage

```
rray_dot(x, y)
```

Arguments

`x, y`	Arrays or rrays that are either 1D or 2D.

Details

Due to some peculiarities with how `%*%` dispatches with S3 objects, calling `%*%` directly with an rray will compute the matrix multiplication correctly, but the class will be lost. `rray_dot()` ensures that the rray class is maintained.

Value

The result of the matrix multiplication of `x` and `y`. See `%*%` for the exact details. The common type of `x` and `y` will be preserved.

Examples

```
rray_dot(1:5, 1:5)

rray_dot(rray(1:5), 1:5)
```

`rray_duplicate` *Find duplicated values in an array*

Description

- `rray_duplicate_any()`: returns a logical with the same shape and type as `x` except over the `axes`, which will be reduced to length 1. This function detects the presence of any duplicated values along the `axes`.
- `rray_duplicate_detect()`: returns a logical with the same shape and type as `x` describing if that element of `x` is duplicated elsewhere.
- `rray_duplicate_id()`: returns an integer with the same shape and type as `x` giving the location of the first occurrence of the value.

Usage

```
rray_duplicate_any(x, axes = NULL)

rray_duplicate_detect(x, axes = NULL)

rray_duplicate_id(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, array, or rray.
`axes`	An integer vector. The default of `NULL` looks for duplicates over all axes.

Value

See the description for return value details.

See Also

`rray_unique()` for functions that work with the dual of duplicated values: unique values.

`vctrs::vec_duplicate_any()` for functions that detect duplicates among any type of vector object.

Examples

```
x <- rray(c(1, 1, 2, 2), c(2, 2))
x <- rray_set_row_names(x, c("r1", "r2"))
x <- rray_set_col_names(x, c("c1", "c2"))

# Are there duplicates along the rows?
rray_duplicate_any(x, 1L)

# Are there duplicates along the columns?
rray_duplicate_any(x, 2L)
```

```
# Create a 3d version of x
# where the columns are not unique
y <- rray_expand(x, 1)

# Along the rows, all the values are unique...
rray_duplicate_any(y, 1L)

# ...but along the columns there are duplicates
rray_duplicate_any(y, 2L)

# ------------------------------------------------------------------------

z <- rray(c(1, 1, 2, 3, 1, 4, 5, 6), c(2, 2, 2))

# rray_duplicate_detect() looks for any
# duplicates along the axes of interest
# and returns `TRUE` wherever a duplicate is found
# (including the first location)
rray_duplicate_detect(z, 1)

# Positions 1 and 5 are the same!
rray_duplicate_detect(z, 3)

# ------------------------------------------------------------------------

# rray_duplicate_id() returns the location
# of the first occurance along each axis.
# Compare to rray_duplicate_detect()!
rray_duplicate_detect(z, 1)
rray_duplicate_id(z, 1)
```

`rray_elems` *Compute the number of elements in an array*

Description

`rray_elems()` computes the number of individual elements in an array. It generally computes the same thing as `length()`, but has a more predictable name.

Usage

```
rray_elems(x)
```

Arguments

`x`	A vector, matrix, array or rray.

Value

A single integer. The number of elements in `x`.

Examples

```
rray_elems(1:5)

rray_elems(matrix(1, 2, 2))

# It is different from `vec_size()`,
# which only returns the number of
# observations
library(vctrs)
vec_size(matrix(1, 2, 2))
```

---

rray_expand *Insert a dimension into an rray*

Description

`rray_expand()` inserts a new dimension at the `axis` dimension. This expands the number of dimensions of `x` by `1`.

Usage

```
rray_expand(x, axis)
```

Arguments

`x`	An rray.
`axis`	An integer of size `1` specifying the location of the new dimension.

Details

Dimension names are kept through the insertion of the new dimension.

Value

`x` with a new dimension inserted at the `axis`.

Examples

```
x <- rray(1:10, c(5, 2))
x <- rray_set_row_names(x, letters[1:5])
x <- rray_set_col_names(x, c("c1", "c2"))

# (5, 2)
# Add dimension to the front
# (1, 5, 2) = 1 row, 5 cols, 2 deep
rray_expand(x, 1)

# (5, 2)
# Add dimension to the middle
# (5, 1, 2) = 5 rows, 1 col, 2 deep
rray_expand(x, 2)

# (5, 2)
# Add dimension to the end
# (5, 2, 1) = 5 rows, 2 cols, 1 deep
rray_expand(x, 3)

# In some cases this is different than a simple
# rray_reshape() because the dimension names
# follow the original dimension position
# - 5 row names follow to the new 5 column position
# - 2 col names follow to the new 2 deep position
# - result has no row names because that is the new axis
rray_expand(x, 1)

# A reshape, on the other hand,
# drops all dimension names
rray_reshape(x, c(1, 5, 2))
```

---

rray_extract<- *Get or set elements of an array by index*

Description

`rray_extract()` is the counterpart to `rray_yank()`. It extracts elements from an array *by index*. It *always* drops dimensions (unlike `rray_subset()`), and a 1D object is always returned.

Usage

```
rray_extract(x, ...) <- value

rray_extract_assign(x, ..., value)

rray_extract(x, ...)
```

Arguments

`x`	A vector, matrix, array, or rray.
`...`	A specification of indices to extract. • Integer-ish indices extract specific elements of dimensions. • Logical indices must be length 1, or the length of the dimension you are subsetting over. • Character indices are only allowed if `x` has names for the corresponding dimension. • `NULL` is treated as `0`.
`value`	The value to assign to the location specified by `...`. Before assignment, `value` is cast to the type and dimension of `x` after extracting elements with `...`.

Details

Like `[[`, `rray_extract()` will *never* keep dimension names.

`rray_extract()` works with base R objects as well.

`rray_extract()` is similar to the traditional behavior of `x[[i, j, ...]]`, but allows each subscript to have length >1.

Value

A 1D vector of elements extracted from `x`.

See Also

Other rray subsetters: `rray_slice<-`, `rray_subset<-`, `rray_yank<-`

Examples

```
x <- rray(1:16, c(2, 4, 2), dim_names = list(c("r1", "r2"), NULL, NULL))

# Extract the first row and flatten it
rray_extract(x, 1)

# Extract the first row and first two columns
rray_extract(x, 1, 1:2)

# You can assign directly to these elements
rray_extract(x, 1, 1:2) <- NA
x
```

`rray_flatten` *Flatten an array*

Description

`rray_flatten()` squashes the dimensionality of `x` so that the result is a 1D object.

Usage

```
rray_flatten(x)
```

Arguments

`x`	A vector, matrix, array or rray.

Details

This function is similar to `as.vector()`, but keeps the class of the object and can keep the dimension names if applicable.

Dimension names are kept using the same rules that would be applied if you would have called `rray_reshape(x,prod(rray_dim(x)))`. Essentially this means that names in the first dimension are kept if `x` is either already a 1D vector, or a higher dimensional object with 1's in all dimensions except for the first one.

Value

A 1D object with the same container type as `x`.

Examples

```
library(magrittr)

x <- rray(1:10, c(5, 2))

rray_flatten(x)

# Dimension names are kept here
# (2) -> (2)
y <- rray(1:2) %>% rray_set_axis_names(1, letters[1:2])
rray_flatten(y)

# And they are kept here
# (2, 1) -> (2)
y_one_col <- rray_reshape(y, c(2, 1))
rray_flatten(y_one_col)

# But not here, since the size of the first dim changes
# (1, 2) -> (2)
y_one_row <- t(y_one_col)
```

```
rray_flatten(y_one_row)
```

`rray_flip` *Flip an rray along a dimension*

Description

`rray_flip()` reverses the elements of an rray along a single dimension.

Usage

```
rray_flip(x, axis)
```

Arguments

`x`	An rray.
`axis`	An integer of size `1` specifying the dimension to flip.

Details

Dimension names are flipped as well.

Value

`x` but with reversed elements along `axis`.

Examples

```
x <- rray(1:10, c(5, 2))
x <- rray_set_row_names(x, letters[1:5])
x <- rray_set_col_names(x, c("c1", "c2"))

rray_flip(x, 1)

rray_flip(x, 2)
```

`rray_full_like` *Create an array like* `x`

Description

- `rray_full_like()` creates an array with the same type and size as `x`, but filled with `value`.
- `rray_ones_like()` is `rray_full_like()` with `value = 1`.
- `rray_zeros_like()` is `rray_full_like()` with `value = 0`.

Usage

```
rray_full_like(x, value)

rray_ones_like(x)

rray_zeros_like(x)
```

Arguments

`x`	A vector, matrix, array or rray.
`value`	A value coercible to the same *inner* type as `x` that will be used to fill the result (If `x` is an integer matrix, then `value` will be coerced to an integer).

Details

No dimension names will be on the result.

Value

An object with the same type as `x`, but filled with `value`.

Examples

```
x <- rray(1:10, c(5, 2))

# Same shape and type as x, but filled with 1
rray_full_like(x, 1L)

# `fill` is coerced to `x` if it can be
rray_full_like(x, FALSE)

# `value = 1`
rray_ones_like(x)

# When logicals are used, it is filled with TRUE
rray_ones_like(c(FALSE, TRUE))
```

```
# `value = 0`
rray_zeros_like(x)
```

`rray_hypot` *Compute the square root of the sum of squares*

Description

`rray_hypot()` computes the elementwise square root of the sum of squares of `x` and `y`.

Usage

```
rray_hypot(x, y)
```

Arguments

`x, y`	A vector, matrix, array or rray.

Value

An object of the common type of `x` and `y` containing the square root of the sum of squares.

Examples

```
x <- matrix(c(2, 4, 6))

# With broadcasting
rray_hypot(x, t(x))
```

`rray_if_else` *Conditional selection*

Description

`rray_if_else()` is like `ifelse()`, but works with matrices and arrays, and fully supports broadcasting between the three inputs. Before the operation is applied, `condition` is cast to a logical, and `true` and `false` are cast to their common type.

Usage

```
rray_if_else(condition, true, false)
```

Arguments

condition	A logical vector, matrix, array of rray.
true	A vector, matrix, array, or rray. This is the value in the result when condition is TRUE.
false	A vector, matrix, array, or rray. This is the value in the result when condition is FALSE.

Details

The dimension names of the output are taken as the common names of true and false.

Examples

```
cond <- c(TRUE, FALSE)

true <- array(
  1:2,
  dimnames = list(c("r1", "r2"))
)

false <- rray(
  c(3, 4, 5, 6),
  dim = c(2, 2),
  dim_names = list(c("rr1", "rr2"), c("c1", "c2"))
)

# - All inputs are broadcast to a common
#   shape of (2, 2).
# - The first row of the output comes from
#   `true`, the second row comes from `false`.
# - The names come from both `true` and `false`.
rray_if_else(cond, true, false)
```

---

rray_max *Calculate the maximum along an axis*

Description

rray_max() computes the maximum along a given axis or axes. The dimensionality of x is retained in the result.

Usage

```
rray_max(x, axes = NULL)
```

Arguments

x	A vector, matrix, or array to reduce.
axes	An integer vector specifying the axes to reduce over. 1 reduces the number of rows to 1, performing the reduction along the way. 2 does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Value

The result of the reduction with the same shape as x, except along axes, which have been reduced to size 1.

See Also

Other reducers: rray_mean, rray_min, rray_prod, rray_sum

Examples

```
x <- rray(1:10, c(5, 2))

rray_max(x)

rray_max(x, 1)

rray_max(x, 2)
```

---

rray_max_pos *Locate the position of the maximum value*

Description

rray_max_pos() returns the integer position of the maximum value over an axis.

Usage

```
rray_max_pos(x, axis = NULL)
```

Arguments

x	A vector, matrix, array, or rray.
axis	A single integer specifying the axis to compute along. 1 computes along rows, reducing the number of rows to 1. 2 does the same, but along columns, and so on for higher dimensions. The default of NULL first flattens x to 1-D.

Value

An integer object of the same type and shape as x, except along axis, which has been reduced to size 1.

Examples

```
x <- rray(c(1:10, 20:11), dim = c(5, 2, 2))

# Find the max position over all of x
rray_max_pos(x)

# Compute along the rows
rray_max_pos(x, 1)

# Compute along the columns
rray_max_pos(x, 2)
```

---

rray_mean *Calculate the mean along an axis*

Description

`rray_mean()` computes the mean along a given axis or axes. The dimensionality of `x` is retained in the result.

Usage

```
rray_mean(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, or array to reduce.
`axes`	An integer vector specifying the axes to reduce over. `1` reduces the number of rows to 1, performing the reduction along the way. `2` does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Value

The result of the reduction as a double with the same shape as `x`, except along `axes`, which have been reduced to size 1.

See Also

Other reducers: `rray_max`, `rray_min`, `rray_prod`, `rray_sum`

Examples

```
x <- rray(1:10, c(5, 2))

rray_mean(x)

rray_mean(x, 1)

rray_mean(x, 2)
```

---

rray_min *Calculate the minimum along an axis*

Description

`rray_min()` computes the minimum along a given axis or axes. The dimensionality of `x` is retained in the result.

Usage

```
rray_min(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, or array to reduce.
`axes`	An integer vector specifying the axes to reduce over. `1` reduces the number of rows to 1, performing the reduction along the way. `2` does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Value

The result of the reduction with the same shape as `x`, except along `axes`, which have been reduced to size 1.

See Also

Other reducers: `rray_max`, `rray_mean`, `rray_prod`, `rray_sum`

Examples

```
x <- rray(1:10, c(5, 2))

rray_min(x)

rray_min(x, 1)

rray_min(x, 2)
```

rray_min_pos *Locate the position of the minimum value*

Description

`rray_min_pos()` returns the integer position of the minimum value over an axis.

Usage

```
rray_min_pos(x, axis = NULL)
```

Arguments

`x`	A vector, matrix, array, or rray.
`axis`	A single integer specifying the axis to compute along. `1` computes along rows, reducing the number of rows to 1. `2` does the same, but along columns, and so on for higher dimensions. The default of `NULL` first flattens `x` to 1-D.

Value

An integer object of the same type and shape as `x`, except along `axis`, which has been reduced to size 1.

Examples

```
x <- rray(c(1:10, 20:11), dim = c(5, 2, 2))

# Flatten x, then find the position of the max value
rray_min_pos(x)

# Compute along the rows
rray_min_pos(x, 1)

# Compute along the columns
rray_min_pos(x, 2)
```

rray_multiply_add *Fused multiply-add*

Description

`rray_multiply_add()` computes `x * y + z`, with broadcasting. It is more efficient than simply doing those operations in sequence.

Usage

```
rray_multiply_add(x, y, z)
```

Arguments

x, y, z	A vector, matrix, array or rray.

Value

An object of the common type of the inputs, containing the result of the multiply-add operation.

Examples

```
rray_multiply_add(2, 3, 5)

# Using broadcasting
rray_multiply_add(matrix(1:5), matrix(1:2, nrow = 1L), 3L)

# ^ Equivalent to:
x <- matrix(rep(1:5, 2), ncol = 2)
y <- matrix(rep(1:2, 5), byrow = TRUE, ncol = 2)
z <- matrix(3L, nrow = 5, ncol = 2)
x * y + z
```

---

rray_prod *Calculate the product along an axis*

Description

`rray_prod()` computes the product along a given axis or axes. The dimensionality of `x` is retained in the result.

Usage

```
rray_prod(x, axes = NULL)
```

Arguments

x	A vector, matrix, or array to reduce.
axes	An integer vector specifying the axes to reduce over. `1` reduces the number of rows to 1, performing the reduction along the way. `2` does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Value

The result of the reduction as a double with the same shape as `x`, except along `axes`, which have been reduced to size 1.

See Also

Other reducers: `rray_max`, `rray_mean`, `rray_min`, `rray_sum`

Examples

```
x <- rray(1:10, c(5, 2))

rray_prod(x)

rray_prod(x, 1)

rray_prod(x, 2)
```

`rray_reshape` *Reshape an array*

Description

`rray_reshape()` is similar to `dim()<-`. It reshapes `x` in such a way that the dimensions are different, but the total size of the array is still the same (as measured by `rray_elems()`).

Usage

```
rray_reshape(x, dim)
```

Arguments

`x`	A vector, matrix, array or rray.
`dim`	An integer vector. The dimension to reshape to.

Value

`x` reshaped to the new dimensions of `dim`.

Examples

```
x <- matrix(1:6, ncol = 1)

# Reshape with the same dimensionality
rray_reshape(x, c(2, 3))

# Change the dimensionality and the dimensions
rray_reshape(x, c(3, 2, 1))

# You cannot reshape to a total size that is
# different from the current size.
```

```
try(rray_reshape(x, c(6, 2)))

# Note that you can broadcast to these dimensions!
rray_broadcast(x, c(6, 2))
```

`rray_rotate` *Rotate an array*

Description

`rray_rotate()` rotates an array along the plane defined by `c(from,to)`. It can be thought of as sequentially rotating the array 90 degrees a set number of `times`.

Usage

```
rray_rotate(x, from = 1, to = 2, times = 1)
```

Arguments

`x`	A matrix, array, or rray.
`from, to`	Single integer values. The direction of the rotation goes from `from` and towards `to`. When using a 2D matrix, this can be thought of as rotating *counter clockwise* starting at `from` and going towards `to`. To go *clockwise*, reverse `from` and `to`.
`times`	A single integer. The number of times to perform the rotation. One of: `1`, `2`, `3`. Or, equivalently: `-3`, `-2`, `-1`.

Details

A rotation can be a hard thing to wrap your head around. I encourage looking at the examples and starting with 2D to try and understand what is happening before moving up to higher dimensions.

Note that a rotation is *not* the same thing as a transpose.

Generally, you can predict the output of rotating using `from` and `to` by switching the dimensions at the `from` and `to` axes position. This gives you the shape of the output. So, a `(5,2,4)` array rotated using `from = 1` and `to = 3` would have a resulting shape of `(4,2,5)`. Note that using `from = 3` and `to = 1` would give the same shape. The "direction" of how these are rotated is controlled by the ordering of `from` and `to`.

Value

`x` rotated along the axis described by `from` and `to`.

Examples

```
# ------------------------------------------------------------------------
# 2D example

x <- rray(1:6, c(3, 2))
x <- rray_set_row_names(x, c("r1", "r2", "r3"))
x <- rray_set_col_names(x, c("c1", "c2"))

# "counter clockwise" rotation turning the
# rows into columns
rray_rotate(x)

# "clockwise" by reversing the direction
rray_rotate(x, from = 2, to = 1)

# Rotate twice (180 degrees)
# Direction doesn't matter here, the following
# give the same result
rray_rotate(x, times = 2)
rray_rotate(x, from = 2, to = 1, times = 2)

# ------------------------------------------------------------------------
# 3D example

x_3d <- rray_expand(x, 3)

# - Rotations on the (1, 3) axis plane
# - Dimensions go from (3, 2, 1) -> (1, 2, 3) in both cases
# - And the direction of how that happens is controlled
#   by `from` and `to`
rray_rotate(x_3d, from = 1, to = 3)

rray_rotate(x_3d, from = 3, to = 1)
```

---

rray_slice<- *Get or set a slice of an array*

Description

`rray_slice()` is a shortcut wrapper around `rray_subset()` that is useful for easily subsetting along a single axis.

Usage

```
rray_slice(x, i, axis) <- value

rray_slice_assign(x, i, axis, value)

rray_slice(x, i, axis)
```

Arguments

x	A vector, matrix, array or rray.
i	Indices to extract along a single axis. • Integer indices extract specific elements of the axis dimension. • Logical indices must be length 1, or the length of the axis dimension. • Character indices are only allowed if x has names for the axis dimension. • NULL is treated as 0.
axis	An integer. The axis to subset along.
value	A value to be assigned to the location at rray_slice(x,i,axis). It will be cast to the type and dimension of the slice of x.

Details

rray_slice() does exactly the same thing as rray_subset(), and is mainly helpful for higher dimensional objects, when you need to subset along, for example, only the 4th dimension.

Value

x with the i elements extracted from the axis.

See Also

Other rray subsetters: rray_extract<-, rray_subset<-, rray_yank<-

Examples

```
x <- rray(1:16, c(2, 2, 2, 2))

# Selecting the first column
rray_slice(x, i = 1, axis = 2)

# rray_slice() is particularly useful for
# subsetting higher dimensions because you don't
# have to worry about the commas
rray_slice(x, i = 2, axis = 4)

# Compare the above with the equivalent using `[`
x[, , , 2]

# `i` can be a character vector if `x` has names along `axis`
x <- rray_set_axis_names(x, axis = 4, c("foo", "bar"))
rray_slice(x, "bar", axis = 4)

# The assignment variation can be useful
# for assigning to higher dimensional elements
rray_slice(x, 1, 3) <- matrix(c(99, 100), nrow = 1)
```

rray_sort *Sort an array*

Description

`rray_sort()` returns an array with the same dimensions as `x`, but sorted along the specified axis.

Usage

```
rray_sort(x, axis = NULL)
```

Arguments

`x`	A vector, matrix, array, or rray.
`axis`	A single integer specifying the axis to compute along. `1` sorts along rows, `2` sorts along columns. The default of `NULL` first flattens `x` to 1-D, sorts, and then reconstructs the original dimensions.

Details

Dimension names are lost along the axis that you sort along. If `axis = NULL`, then all dimension names are lost. In both cases, meta names are kept. The rationale for this is demonstrated in the examples. There, when you sort `y` along the rows, the rows in the first column change position, but the rows in the second column do not, so there is no rational order that the row names can be placed in.

Value

An object with the same dimensions as `x`, but sorted along `axis`. The dimension names will be lost along the axis you sort along.

Examples

```
x <- rray(c(20:11, 1:10), dim = c(5, 2, 2))

# Flatten, sort, then reconstruct the shape
rray_sort(x)

# Sort, looking along the rows
rray_sort(x, 1)

# Sort, looking along the columns
rray_sort(x, 2)

# Sort, looking along the third dimension
# This switches the 20 with the 1, the
# 19 with the 2, and so on
rray_sort(x, 3)
```

```
# ------------------------------------------------------------------------
# Dimension names

y <- rray(
  c(2, 1, 1, 2),
  dim = c(2, 2),
  dim_names = list(
    r = c("r1", "r2"),
    c = c("c1", "c2")
  )
)

# Dimension names are dropped along the axis you sort along
rray_sort(y, 1)
rray_sort(y, 2)

# All dimension names are dropped if `axis = NULL`
rray_sort(y)
```

`rray_split` *Split an array along axes*

Description

`rray_split()` splits `x` into equal pieces, splitting along the `axes`.

Usage

```
rray_split(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, array, or rray.
`axes`	An integer vector. The axes to split on. The default splits along all axes.

Details

`rray_split()` works by splitting along the `axes`. The result is a list of sub arrays, where the `axes` of each sub array have been reduced to length 1. This is consistent with how reducers like `rray_sum()` work. As an example, splitting a `(2,3,5)` array along `axes = c(2,3)` would result in a list of 15 (from `3 * 5`) sub arrays, each with shape `(2,1,1)`.

Value

A list of sub arrays of type `x`.

Examples

```
x <- matrix(1:8, ncol = 2)

# Split the rows
# (4, 2) -> (1, 2)
rray_split(x, 1)

# Split the columns
# (4, 2) -> (4, 1)
rray_split(x, 2)

# Split along multiple dimensions
# (4, 2) -> (1, 1)
rray_split(x, c(1, 2))

# The above split is the default behavior
rray_split(x)

# You can technically split with a size 0 `axes`
# argument, which essentially requests no axes
# to be split and is the same as `list(x)`
rray_split(x, axes = integer(0))

# ---------------------------------------------------------------------------
# 4 dimensional example

x_4d <- rray(
  x = 1:16,
  dim = c(2, 2, 2, 2),
  dim_names = list(
    c("r1", "r2"),
    c("c1", "c2"),
    c("d1", "d2"),
    c("e1", "e2")
  )
)

# Split along the 1st dimension (rows)
# (2, 2, 2, 2) -> (1, 2, 2, 2)
rray_split(x_4d, 1)

# Split along columns
# (2, 2, 2, 2) -> (2, 1, 2, 2)
rray_split(x_4d, 2)

# Probably the most useful thing you might do
# is use this to split the 4D array into a set
# of 4 2D matrices.
rray_split(x_4d, c(3, 4))
```

rray_squeeze *Squeeze an rray*

Description

`rray_squeeze()` is conceptually similar to `base::drop()`, but it allows for the specification of specific dimensions to squeeze.

Usage

```
rray_squeeze(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, array or rray.
`axes`	An integer vector specifying the size 1 dimensions to drop. If `NULL`, all size 1 dimensions are dropped.

Details

The dimension name handling of `rray_squeeze()` is essentially identical to `drop()`, but some explanation is always helpful:

- Dimension names are removed from the axes that are squeezed. So squeezing a `(2,1,2)` object results in a `(2,2)` object using the dimension names from the original first and third dimensions.
- When all dimensions are squeezed, as in the case of `(1,1,1)`, then the first dimension names that are found are the ones that are used in the `(1)` result.

Value

`x` with the `axes` dropped, if possible.

Examples

```
# (10, 1) -> (10)
x <- rray(1:10, c(10, 1))
rray_squeeze(x)

# Multiple squeezed dimensions
# (10, 1, 1) -> (10)
y <- rray_reshape(x, c(10, 1, 1))
rray_squeeze(y)

# Use `axes` to specify dimensions to drop
# (10, 1, 1) -> drop 2 -> (10, 1)
rray_squeeze(y, axes = 2)
```

```
# Dimension names are kept here
# (10, 1) -> (10)
x <- rray_set_row_names(x, letters[1:10])
rray_squeeze(x)

# And they are kept here
# (1, 10) -> (10)
rray_squeeze(t(x))
```

`rray_subset<-` *Get or set dimensions of an array*

Description

`rray_subset()` extracts dimensions from an array *by index*. It powers `[` for rray objects. Notably, it *never* drops dimensions, and ignores trailing commas.

Usage

```
rray_subset(x, ...) <- value

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

rray_subset_assign(x, ..., value)

rray_subset(x, ...)

## S3 method for class 'vctrs_rray'
x[..., drop = FALSE]
```

Arguments

`x`	A vector, matrix, array, or rray.
`...`	A specification of indices to extract. • Integer-ish indices extract specific elements of dimensions. • Logical indices must be length 1, or the length of the dimension you are subsetting over. • Character indices are only allowed if `x` has names for the corresponding dimension. • `NULL` is treated as `0`.
`value`	The value to assign to the location specified by `...`. Before assignment, `value` is cast to the type and dimension of `x[...]`.
`drop`	Ignored, but preserved for better error messages with code that might have used arrays before.

Details

`rray_subset()` and its assignment variant can also be used with base R matrices and arrays to get rray subsetting behavior with them.

Value

`x` subset by the specification defined in the `....`

The assignment variants return `x` modified by having the elements of `value` inserted into the positions defined by `....`

Differences from base R

- `rray_subset()` *never* drops dimensions.
- `rray_subset()` ignores trailing commas. This has the nice property of making `x[1] == x[1,]`.
- `rray_subset()<-` casts `value` to `x`, rather than casting `x` to `value`.

See Also

`pad()`

Other rray subsetters: `rray_extract<-`, `rray_slice<-`, `rray_yank<-`

Examples

```
x <- rray(1:8, c(2, 2, 2))

# `rray_subset()` powers `[` so these are identical
rray_subset(x, 1)
x[1]

# Trailing dots are ignored, so these are identical
x[1]
x[1,]

# Missing arguments are treated as selecting the
# entire dimension, consistent with base R.
# This selects all of the rows, and the first column.
x[,1]

# Notice that you can't actually do the above with base
# R. It requires you to fully specify the dimensions of `x`.
# This would throw an error.
x_arr <- as_array(x)
try(x_arr[,1])

# To get the same behavior, you have to do:
x_arr[, 1, , drop = FALSE]

# Note that you can use base R arrays with `rray_subset()`
rray_subset(x_arr, , 1)
```

```
# For higher dimensional objects, `pad()` can be
# useful for automatically adding commas. The
# following are equivalent:
x[pad(), 1]
x[, , 1]

# You can assign to index locations with
# `x[...] <- value`
# This assigns 99 to the entire first row
x[1] <- 99
x

# First row in the first
# element of the 3rd dimension
x[1, , 1] <- 100
x

# Note that `value` is broadcast to the shape
# of `x[...]`. So this...
x[,1] <- matrix(5)

# ...becomes the same as
x[,1] <- array(5, c(2, 1, 2))

# You can also use `rray_subset<-()` directly to
# use these semantics with base R
rray_subset(x_arr, , 1) <- matrix(5)
x_arr
```

`rray_sum` *Calculate the sum along an axis*

Description

`rray_sum()` computes the sum along a given axis or axes. The dimensionality of `x` is retained in the result.

Usage

```
rray_sum(x, axes = NULL)
```

Arguments

`x`	A vector, matrix, or array to reduce.
`axes`	An integer vector specifying the axes to reduce over. `1` reduces the number of rows to 1, performing the reduction along the way. `2` does the same, but with the columns, and so on for higher dimensions. The default reduces along all axes.

Value

The result of the reduction as a double with the same shape as `x`, except along `axes`, which have been reduced to size 1.

See Also

Other reducers: `rray_max`, `rray_mean`, `rray_min`, `rray_prod`

Examples

```
x <- rray(1:10, c(5, 2))

# Reduce the number of rows to 1,
# summing along the way
rray_sum(x, 1)

# Reduce the number of columns to 1,
# summing along the way
rray_sum(x, 2)

# Reduce along all axes, but keep dimensions
rray_sum(x)

# Column-wise proportions
x / rray_sum(x, 1)

# Row-wise proportions
x / rray_sum(x, 2)

# Reducing over multiple axes
# This reduces over the rows and columns
# of each mini-matrix in the 3rd dimension
y <- rray(1:24, c(2, 3, 4))
rray_sum(y, c(1, 2))
```

`rray_tile` *Tile an array*

Description

Tile an array

Usage

```
rray_tile(x, times)
```

Arguments

x	A vector, matrix, array or rray.
times	An integer vector. The number of times to repeat the array along an axis.

Details

`rray_tile()` should not be used as a replacement for `rray_broadcast()`, as it is generally less efficient.

Value

`x` with dimensions repeated as described by `times`.

Examples

```
x <- matrix(1:5)

# Repeat the rows twice
rray_tile(x, 2)

# Repeat the rows twice and the columns three times
rray_tile(x, c(2, 3))

# Tile into a third dimension
rray_tile(x, c(1, 2, 2))
```

`rray_transpose` *Transpose an array*

Description

`rray_transpose()` transposes `x` along axes defined by `permutation`. By default, a standard transpose is performed, which is equivalent to permuting along the reversed dimensions of `x`.

Usage

```
rray_transpose(x, permutation = NULL)

## S3 method for class 'vctrs_rray'
t(x)
```

Arguments

x	A vector, matrix, array, or rray.
permutation	This should be some permutation of `1:n` with `n` being the number of dimensions of `x`. If `NULL`, the reverse of `1:n` is used, which is the normal transpose.

Details

Unlike `t()`, using `rray_transpose()` on a vector does not transpose it, as it is a 1D object, and the consistent result of transposing a 1D object is itself.

`t.vctrs_rray()` uses the base R's `t()` behavior to be consistent with user expectations about transposing 1D objects.

There is an `aperm()` method for `rray` objects as well. Unlike base R, it currently does not accept character strings for `perm`.

Value

`x` transposed along the axes defined by the `permutation`.

Examples

```
x <- rray(
 1:6,
 c(3, 2),
 dim_names = list(rows = c("r1", "r2", "r3"), cols = c("c1", "c2"))
)

# A standard transpose
rray_transpose(x)

# Identical to
rray_transpose(x, rev(1:2))

x_3d <- rray_broadcast(x, c(3, 2, 2))

# transpose here is like setting
# `permutation = c(3, 2, 1)`
# so the result should change _shape_ like:
# (3, 2, 2) -> (2, 2, 3)
rray_transpose(x_3d)

# This transposes the "inner" matrices
# (flips the first and second dimension)
# and leaves the 3rd dimension alone
rray_transpose(x_3d, c(2, 1, 3))

# ------------------------------------------------------------------------------
# Difference from base R

# Coerces 1:5 into a 2D matrix, then transposes
t(1:5)

# Leaves it as a 1D array and does nothing
rray_transpose(1:5)

# t.vctrs_rray() has the same behavior
# as base R
```

```
t(rray(1:5))
```

`rray_unique` *Find and count unique values in an array*

Description

- `rray_unique()`: the unique values.
- `rray_unique_loc()`: the locations of the unique values.
- `rray_unique_count()`: the number of unique values.

Usage

```
rray_unique(x, axis)

rray_unique_loc(x, axis)

rray_unique_count(x, axis)
```

Arguments

`x`	A vector, matrix, array, or rray.
`axis`	A single integer. The axis to index `x` by.

Details

The family of unique functions work in the following manner:

1. `x` is split into pieces using the `axis` as the dimension to index along.
2. Each of those pieces is flattened to 1D.
3. The uniqueness test is done between those flattened pieces and the final output is restored from that result.

As an example, if `x` has dimensions of `(2,3,2)` and `axis = 2`, then you can think of `x` as being broken into `x[,1]`, `x[,2]` and `x[,3]`. Each of those three pieces are then flattened, and a vctrs unique function is called on the list of those flattened inputs.

The result of calling `rray_unique()` will always have the same dimensions as `x`, except along `axis`, which is allowed to be less than the original axis size if any duplicate entries are removed.

Unlike the duplicate functions, the unique functions only take a singular `axis` argument, rather than `axes`. The reason for this is that if the unique functions were defined in any other way, they would allow for *ragged arrays*, which are not defined in rray.

When duplicates are detected, the *first* unique value is used in the result.

Value

- `rray_unique()`: an array the same type as `x` containing only unique values. The dimensions of the return value are the same as `x` except on the `axis`, which might be smaller than the original dimension size if any duplicate entries were removed.
- `rray_unique_loc()`: an integer vector, giving locations of the unique values.
- `rray_unique_count()`: an integer vector of length 1, giving the number of unique values.

See Also

`rray_duplicate_any()` for functions that work with the dual of unique values: duplicated values.

`vctrs::vec_unique()` for functions that detect unique values among any type of vector object.

Examples

```
x_dup_rows <- rray(c(1, 1, 3, 3, 2, 2, 4, 4), c(2, 2, 2))
x_dup_rows <- rray_set_row_names(x_dup_rows, c("r1", "r2"))
x_dup_rows <- rray_set_col_names(x_dup_rows, c("c1", "c2"))

# Duplicate rows
# `x_dup_rows[1] == x_dup_rows[2]`
rray_unique(x_dup_rows, 1)

# Duplicate cols
# `x_dup_cols[, 1] == x_dup_cols[, 2]`
x_dup_cols <- rray_transpose(x_dup_rows, c(2, 1, 3))
rray_unique(x_dup_cols, 2)

# Duplicate 3rd dim
# `x_dup_layers[, , 1] == x_dup_layers[, , 2]`
x_dup_layers <- rray_transpose(x_dup_rows, c(2, 3, 1))
rray_unique(x_dup_layers, 3)

# rray_unique_loc() returns an
# integer vector you can use
# to subset out the unique values along
# the axis you are interested in
x_dup_cols[, rray_unique_loc(x_dup_cols, 2L)]

# Only 1 unique column
rray_unique_count(x_dup_cols, 2L)

# But 2 unique rows
rray_unique_count(x_dup_cols, 1L)
```

`rray_yank<-` *Get or set elements of an array* by position

Description

- `rray_yank()` gets and sets elements from an array *by position*. It is the complement to `rray_extract()`, which gets and sets elements *by index*.
- For rrays, `[[` is powered by `rray_yank()`, and allows you to select multiple elements by position with the syntax: `x[[i]]`.
- There are three assignment variations that all work essentially the same. They ensures that `value` has the same inner type as `x`, and `value` must be 1D.
  - `rray_yank(x,i) <-value`
  - `x[[i]] <-value`
  - `rray_yank_assign(x,i,value)`

Usage

```
rray_yank(x, i) <- value

## S3 replacement method for class 'vctrs_rray'
x[[i, ...]] <- value

rray_yank_assign(x, i, value)

rray_yank(x, i)

## S3 method for class 'vctrs_rray'
x[[i, ...]]
```

Arguments

`x`	A vector, matrix, array or rray.
`i`	One of the following: • An integer vector specifying the positions of the elements to yank. • A 1D logical vector of either length 1 or `rray_elems(x)`. • A logical with the exact same dimensions as `x`.
`value`	A 1D value to be assigned to the location yanked by `i`. It will be cast to the type of `rray_yank(x,i)`.
`...`	Not used. An error is thrown if extra arguments are supplied here.

Details

`rray_yank()` is meant as a replacement for the traditional behavior of `x[i]`, which extracts multiple elements by their position, and returns a 1D vector. rray is much stricter, and for rrays `x[i]` would return the `i` rows of `x`, without dropping any dimensions. Separating this special behavior of extracting by position into a new function is less surprising, and allows `[` to be more consistent and stable.

`rray_yank()` never keeps dimension names. For >1D objects, this would not be well defined to begin with, so the decision was made to keep this behavior for 1D objects as well. Think of `rray_yank()` as a way to rip out the inner elements of `x`. The dimension names and outer type are not a part of this information.

Value

A 1D vector of elements yanked out of `x`.

The Double Bracket `[[`

`rray_yank()` powers `[[` for rray objects. It works a bit differently from base R. As with `[`, base R allows `[[` to perform two roles. It can extract *one* element by position with `x[[i]]`, or *one* element by index with `x[[i, j, ...]]`. This felt too flexible, so with rray objects `[[` is directly powered by `rray_yank()`, meaning it can only do `x[[i]]`. However, multiple values of `i` are allowed, rather than just 1, meaning that `x[[c(3, 5)]]` will extract the 3rd and 5th positions in `x` and return them as a 1D array.

Notably this means that the index extraction behavior of `x[[i, j, ...]]` is missing with `[[` for rrays. If you want that behavior, see `rray_extract()`.

See Also

Other rray subsetters: `rray_extract<-`, `rray_slice<-`, `rray_subset<-`

Examples

```
x <- rray(10:17, c(2, 2, 2))

# Resulting dimension is always 1D, and is a base R array
rray_yank(x, 1:3)

# Subsetting with a logical is possible if it is either
# length 1 or the length of `x`
rray_yank(x, FALSE)
rray_yank(x, rep(c(TRUE, FALSE), times = rray_elems(x) / 2))

# You can assign a 1D vector to these yanked selections
# Length 1 values are recycled as required
rray_yank(x, c(1, 3, 5)) <- 9

# `rray_yank()` powers `[[` as well
# Notably, you can yank multiple values in `[[`
x[[c(1, 3, 5)]] <- NA
```

```
# Logicals with the same dim as `x` can also be used as a yank indexer
# This comes in handy as a way to remove NA values
x[[is.na(x)]] <- 0
```

`vec_arith.vctrs_rray` *vctrs compatibility functions*

Description

These functions are the extensions that allow rray objects to work with vctrs.

Usage

```
## S3 method for class 'vctrs_rray'
vec_arith(op, x, y)

## S3 method for class 'vctrs_rray_dbl'
vec_ptype2(x, y, ...)

## S3 method for class 'vctrs_rray_int'
vec_ptype2(x, y, ...)

## S3 method for class 'vctrs_rray_lgl'
vec_ptype2(x, y, ...)

## S3 method for class 'vctrs_rray_int'
vec_cast(x, to, ...)

## S3 method for class 'vctrs_rray_dbl'
vec_cast(x, to, ...)

## S3 method for class 'vctrs_rray_lgl'
vec_cast(x, to, ...)
```

Arguments

`op`	An arithmetic operator as a string.
`x, y`	Objects.
`...`	Used to pass along error message information.
`to`	Type to cast to.

Value

See the corresponding vctrs function for the exact return value.