Package 'vctrs'

November 29, 2018

Title Vector Helpers

Version 0.1.0

Description Defines new notions of prototype and size that are used to provide tools for consistent and well-founded type-coercion and size-recycling, and are in turn connected to ideas of type- and size-stability useful for analyzing function interfaces.

License GPL-3

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

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

Depends R (>= 3.1)

Imports backports, digest, glue, rlang, zeallot

Suggests covr, generics, knitr, pillar, pkgdown, rmarkdown, testthat, tibble

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

LazyData true

RoxygenNote 6.1.1

NeedsCompilation yes

Author Hadley Wickham [aut, cre], RStudio [cph]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2018-11-29 17:10:03 UTC

R topics documented:

`dims` *Vector dimensions*

Description

- `vec_empty()` returns `TRUE` if `vec_size()` is zero.
- `vec_dims()` gives the dimensionality (i.e. number of dimensions)
- `vec_dim()` returns the size of each dimension

Usage

```
vec_empty(x)

vec_dim(x)

vec_dims(x)
```

Arguments

`x`	A vector

Details

Unlike base R, we treat vectors with `NULL` dimensions as 1d. This simplifies the type system by eliding a special case. Compared to base R equivalent, `vec_dim()` returns `length()`, not `NULL`, when `x` is 1d.

Examples

```
# Compared to base R
x <- 1:5
dim(x)
vec_dim(x)
```

---

list_of list_of *S3 class for homogenous lists*

Description

A list_of object is a list where each element has the same type. Modifying the list with $, [, and [[preserves the constraint by coercing all input items.

Usage

```
list_of(..., .ptype = NULL)

as_list_of(x, ...)

is_list_of(x)

## S3 method for class 'vctrs_list_of'
vec_type2(x, y)

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

Arguments

...	Vectors to coerce.
.ptype	If NULL, the default, the output type is determined by computing the common type across all elements of .... Alternatively, you can supply .ptype to give the output known type. If getOption("vctrs.no_guessing is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.
x	For as_list_of(), a vector to be coerced to list_of.
y, to	Arguments to vec_type2() and vec_cast().

Examples

```
x <- list_of(1:3, 5:6, 10:15)
if (requireNamespace("tibble", quietly = TRUE)) {
  tibble::tibble(x = x)
}

vec_c(list_of(1, 2), list_of(FALSE, TRUE))
```

partial_frame *Partially specify columns of a data frame*

Description

This special class can be passed to `.ptype` in order to specify the types of only some of the columns in a data frame.

Usage

```
partial_frame(...)
```

Arguments

`...`	Attributes of subclass

Examples

```
pf <- partial_frame(x = double())
pf

vec_rbind(
  data.frame(x = 1L, y = "a"),
  data.frame(x = FALSE, z = 10),
  .ptype = partial_frame(x = double(), a = character())
)
```

vec_assert *Assert an argument has known prototype and/or size*

Description

Assert an argument has known prototype and/or size

Usage

```
vec_assert(x, ptype = NULL, size = NULL)
```

Arguments

`x`	A vector argument to check.
`ptype`	Prototype to compare against.
`size`	Size to compare against

Value

Either an error, or `x`, invisibly.

vec_bind *Combine many data frames into one data frame*

Description

This pair of functions binds together data frames (and vectors), either row-wise or column-wise. Row-binding creates a data frame with common type across all arguments. Column-binding creates a data frame with common length across all arguments.

Usage

```
vec_rbind(..., .ptype = NULL)

vec_cbind(..., .ptype = NULL, .size = NULL)
```

Arguments

`...`	Data frames or vectors. `vec_rbind()` ignores names. `vec_cbind()` preserves outer names, combining with inner names if also present. `NULL` inputs are silently ignored. Empty (e.g. zero row) inputs will not appear in the output, but will affect the derived `.ptype`.
`.ptype`	If `NULL`, the default, the output type is determined by computing the common type across all elements of `...`. Alternatively, you can supply `.ptype` to give the output known type. If `getOption("vctrs.no_guessing` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.
`.size`	If, `NULL`, the default, will determing the number of rows in `vec_cbind()` output by using the standard recycling rules. Alternatively, specify the desired number of rows, and any inputs of length 1 will be recycled appropriately.

Value

A data frame, or subclass of data frame.

If `...` is a mix of different data frame subclases, `vec_type2()` will be used to determine the output type. For `vec_rbind()`, this will determine the type of the container and the type of each column; for `vec_cbind()` it only determines the type of the output container. If there are no non-`NULL` inputs, the result will be `data.frame()`.

Invariants

- `vec_size(vec_rbind(x, y)) == vec_size(x) + vec_size(y)`
- `vec_type(vec_rbind(x, y)) = vec_type_common(x, y)`
- `vec_size(vec_cbind(x, y)) == vec_size_common(x, y)`
- `vec_type(vec_cbind(x, y)) == vec_cbind(vec_type(x), vec_type(x))`

See Also

vec_c() for combining 1d vectors.

Examples

```
# row binding ----------------------------------------

# common columns are coerced to common class
vec_rbind(
  data.frame(x = 1),
  data.frame(x = FALSE)
)

# unique columns are filled with NAs
vec_rbind(
  data.frame(x = 1),
  data.frame(y = "x")
)

# null inputs are ignored
vec_rbind(
  data.frame(x = 1),
  NULL,
  data.frame(x = 2)
)

# bare vectors are treated as rows
vec_rbind(
  c(x = 1, y = 2),
  c(x = 3)
)

# default names will be supplied if arguments are not named
vec_rbind(
  1:2,
  1:3,
  1:4
)

# column binding -------------------------------------

# each input is recycled to have common length
vec_cbind(
  data.frame(x = 1),
  data.frame(y = 1:3)
)

# bare vectors are treated as columns
vec_cbind(
  data.frame(x = 1),
  y = letters[1:3]
)
```

```
# outer names are combined with inner names
vec_cbind(
  x = data.frame(a = 1, b = 2),
  y = 1
)

# duplicate names are flagged
vec_cbind(x = 1, x = 2)
```

---

vec_c *Combine many vectors into one vector*

Description

Combine all arguments into a new vector of common type.

Usage

```
vec_c(..., .ptype = NULL)
```

Arguments

`...`	Vectors to coerce.
`.ptype`	If `NULL`, the default, the output type is determined by computing the common type across all elements of `...`. Alternatively, you can supply `.ptype` to give the output known type. If `getOption("vctrs.no_guessing` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.

Value

A vector with class given by `.ptype`, and length equal to the sum of the `vec_size()` of the contents of `...`.

The vector will have names if the individual components have names (inner names) or if the arguments are named (outer names). If both inner and outer names are present, they are combined with a `..`

Invariants

- `vec_size(vec_c(x, y)) == vec_size(x) + vec_size(y)`
- `vec_type(vec_c(x, y)) == vec_type_common(x, y)`.

See Also

`vec_cbind()`/`vec_rbind()` for combining data frames by rows or columns.

Examples

```
vec_c(FALSE, 1L, 1.5)
vec_c(FALSE, 1L, "x", .ptype = character())

# Date/times --------------------------
c(Sys.Date(), Sys.time())
c(Sys.time(), Sys.Date())

vec_c(Sys.Date(), Sys.time())
vec_c(Sys.time(), Sys.Date())

# Factors -----------------------------
c(factor("a"), factor("b"))
vec_c(factor("a"), factor("b"))
```

---

vec_compare *Compare two vectors*

Description

Compare two vectors

Usage

```
vec_compare(x, y, na_equal = FALSE, .ptype = NULL)
```

Arguments

x, y	Vectors with compatible types and lengths.
na_equal	Should NA values be considered equal?
.ptype	Override to optionally specify common type

Value

An integer vector with values -1 for `x < y`, 0 if `x == y`, and 1 if `x > y`. If `na_equal` is `FALSE`, the result will be `NA` if either `x` or `y` is `NA`.

S3 dispatch

`vec_compare()` is not generic for performance; instead it uses `vec_proxy_compare()` to

Examples

```
vec_compare(c(TRUE, FALSE, NA), FALSE)
vec_compare(c(TRUE, FALSE, NA), FALSE, na_equal = TRUE)

vec_compare(1:10, 5)
vec_compare(runif(10), 0.5)
vec_compare(letters[1:10], "d")

df <- data.frame(x = c(1, 1, 1, 2), y = c(0, 1, 2, 1))
vec_compare(df, data.frame(x = 1, y = 1))
```

---

vec_count *Count unique values in a vector*

Description

Count the number of unique values in a vector. `vec_count()` has two important differences to `table()`: it returns a data frame, and when given multiple inputs (as a data frame), it only counts combinations that appear in the input.

Usage

```
vec_count(x, sort = c("count", "key", "location", "none"))
```

Arguments

`x`	A vector (including a data frame).
`sort`	One of "count", "key", "location", or "none". • "count", the default, puts most frequent values at top • "key", orders by the output key column (i.e. unique values of `x`) • "location", orders by location where key first seen. This is useful if you want to match the counts up to other unique/duplicated functions. • "none", leaves unordered.

Value

A data frame with columns `key` (same type as `x`) and `count` (an integer vector).

Examples

```
vec_count(mtcars$vs)
vec_count(iris$Species)

# If you count a data frame you'll get a data frame
# column in the output
str(vec_count(mtcars[c("vs", "am")]))
```

```
# Sorting --------------------------------------

x <- letters[rpois(100, 6)]
# default is to sort by frequency
vec_count(x)

# by can sort by key
vec_count(x, sort = "key")

# or location of first value
vec_count(x, sort = "location")
head(x)

# or not at all
vec_count(x, sort = "none")
```

`vec_data` *Extract underlying data*

Description

Extract the data underlying an S3 vector object, i.e. the underlying atomic vector or list. Currently, due to the underlying memory architecture of R, this creates a full copy of the underlying data.

Usage

```
vec_data(x)
```

Arguments

`x`	A vector

Value

The data underlying `x`, free from an attributes.

See Also

See `vec_restore()` for the inverse operation: it restores attributes given a bare vector and a prototype; `vec_restore(vec_data(x), x)` will always yield `x`.

vec_duplicate *Find duplicated values*

Description

- `vec_duplicate_any()`: detects the presence of any duplicated values, in the same way as `anyDuplicated()`.
- `vec_duplicate_detect()`: returns a logical vector describing if each element of the vector is duplicated elsewhere. Unlike `duplicated()`, it reports all duplicated values, not just the second and subsequent repetitions.
- `vec_duplicate_id()`: returns an integer vector given the location of the first occurence of the value

Usage

```
vec_duplicate_any(x)

vec_duplicate_detect(x)

vec_duplicate_id(x)
```

Arguments

`x`	A vector (including a data frame).

Value

- `vec_duplicate_any()`: a logical vector of length 1.
- `vec_duplicate_detect()`: a logical vector the same length as `x`
- `vec_duplicate_id()`: an integer vector the same length as `x`

Missing values

In most cases, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. This behaviour would be unappealing here, so these functions consider all `NA`s to be equal. (Similarly, all `NaN` are also considered to be equal.)

Performance

These functions are currently slightly slower than their base equivalents. This is primarily because they do a little more checking and coercion in R, which makes them both a litter safer and more generic. Additionally, the C code underlying vctrs has not yet been implemented: we expect some performance improvements when that happens.

See Also

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

Examples

```
vec_duplicate_any(1:10)
vec_duplicate_any(c(1, 1:10))

x <- c(10, 10, 20, 30, 30, 40)
vec_duplicate_detect(x)
# Note that `duplicated()` doesn't consider the first instance to
# be a duplicate
duplicated(x)

# Identify elements of a vector by the location of the first element that
# they're equal to:
vec_duplicate_id(x)
# Location of the unique values:
vec_unique_loc(x)
# Equivalent to `duplicated()`:
vec_duplicate_id(x) == seq_along(x)
```

---

vec_equal *Test if two vectors are equal*

Description

Test if two vectors are equal

Usage

```
vec_equal(x, y, na_equal = FALSE, .ptype = NULL)

vec_equal_na(x)
```

Arguments

x	Vectors with compatible types and lengths.
y	Vectors with compatible types and lengths.
na_equal	Should NA values be considered equal?
.ptype	Override to optionally specify common type

Value

A logical vector. Will only contain NAs if na_equal is FALSE.

Examples

```
vec_equal(c(TRUE, FALSE, NA), FALSE)
vec_equal(c(TRUE, FALSE, NA), FALSE, na_equal = TRUE)
vec_equal_na(c(TRUE, FALSE, NA))

vec_equal(5, 1:10)
vec_equal("d", letters[1:10])

df <- data.frame(x = c(1, 1, 2, 1), y = c(1, 2, 1, NA))
vec_equal(df, data.frame(x = 1, y = 2))
vec_equal_na(df)
```

`vec_match` *Find matching observations across vectors*

Description

`vec_in()` returns a logical vector based on whether `needle` is found in haystack. `vec_match()` returns an integer vector giving location of `needle` in `haystack`, or `NA` if it's not found.

Usage

```
vec_match(needles, haystack)

vec_in(needles, haystack)
```

Arguments

`needles, haystack`

Vector of `needles` to search for in vector haystack. `haystack` should usually be unique; if not `vec_match()` will only return the location of the first match.

`needles` and `haystack` are coerced to the same type prior to comparison.

Details

`vec_in()` is equivalent to %in%; `vec_match()` is equivalen to `match()`.

Value

A vector the same length as `needles`. `vec_in()` returns a logical vector; `vec_match()` returns an integer vector.

Missing values

In most cases, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. This behaviour would be unappealing here, so these functions consider all `NA`s to be equal. (Similarly, all `NaN` are also considered to be equal.)

Performance

These functions are currently slightly slower than their base equivalents. This is primarily because they do a little more checking and coercion in R, which makes them both a litter safer and more generic. Additionally, the C code underlying vctrs has not yet been implemented: we expect some performance improvements when that happens.

Examples

```
hadley <- strsplit("hadley", "")[[1]]
vec_match(hadley, letters)

vowels <- c("a", "e", "i", "o", "u")
vec_match(hadley, vowels)
vec_in(hadley, vowels)

# Only the first index of duplicates is returned
vec_match(c("a", "b"), c("a", "b", "a", "b"))
```

---

vec_na *Create a missing vector*

Description

Create a missing vector

Usage

```
vec_na(x, n = 1L)
```

Arguments

x	Template of missing vector
n	Desired size of result

Examples

```
vec_na(1:10, 3)
vec_na(Sys.Date(), 5)
vec_na(mtcars, 2)
```

vec_recycle *Vector recycling*

Description

`vec_recycle(x, size)` recycles a single vector to given size. `vec_recycle_common(...)` recycles multiple vectors to their common size. All functions obey the vctrs recycling rules, described below, and will throw an error if recycling is not possible. See `vec_size()` for the precise definition of size.

Usage

```
vec_recycle(x, size)

vec_recycle_common(..., .size = NULL)
```

Arguments

`x, ...`	Vectors to recycle.
`size, .size`	Desired output size. If omitted in `vec_recycle_common()` will use the common size from `vec_size_common()`.

Recycling rules

The common size of two vectors defines the recycling rules, and can be summarise with the following table:

(Note `NULL`s are handled specially; they are treated like empty arguments and hence don't affect the size)

This is a stricter set of rules than base R, which will usually return output of length `max(nx, ny)`, warning if the length of the longer vector is not an integer multiple of the length of the shorter.

We say that two vectors have **compatible size** if they can be recycled to be the same length.

Examples

```
# Inputs with 1 observation are recycled
vec_recycle_common(1:5, 5)
## Not run:
vec_recycle_common(1:5, 1:2)

## End(Not run)

# Inputs with 0 observations
vec_recycle_common(1:5, integer())

# Data frames and matrices are recycled along their rows
vec_recycle_common(data.frame(x = 1), 1:5)
vec_recycle_common(array(1:2, c(1, 2)), 1:5)
vec_recycle_common(array(1:3, c(1, 3, 1)), 1:5)
```

vec_size *Number of observations*

Description

vec_size(x) returns the size of a vector. This is distinct from the length() of a vector because it generalises to the "number of observations" for 2d structures, i.e. it's the number of rows in matrix or a data frame. This definition has the important property that every column of a data frame (even data frame and matrix columns) have the same size. vec_size_common(...) returns the common size of multiple vectors.

Usage

```
vec_size(x)

vec_size_common(..., .size = NULL)
```

Arguments

x, ...	Vector inputs
.size	If NULL, the default, the output size is determined by recycling the lengths of all elements of .... Alternatively, you can supply .size to force a known size.

Details

There is no vctrs helper that retrieves the number of columns: as this is a property of the type.

vec_size() is equivalent to NROW() but has a name that is easier to pronounce, and throws an error when passed non-vector inputs.

Value

An integer (or double for long vectors). Will throw an error if x is not a vector.

vec_size_common() will return NULL if all inputs are NULL or absent.

Invariants

- vec_size(dataframe) == vec_size(dataframe[[i]])
- vec_size(matrix) == vec_size(matrix[, i, drop = FALSE])
- vec_size(vec_c(x, y)) == vec_size(x) + vec_size(y)

See Also

vec_slice() for a variation of [compatible with vec_size(), and vec_recycle() to recycle vectors to common length.

Examples

```
vec_size(1:100)
vec_size(mtcars)
vec_size(array(dim = c(3, 5, 10)))

vec_size(NULL)
# Because vec_size(vec_c(NULL, x)) ==
#   vec_size(NULL) + vec_size(x) ==
#   vec_size(x)

vec_size_common(1:10, 1:10)
vec_size_common(1:10, 1)
vec_size_common(1:10, integer())
```

---

vec_type *Find the prototype of a set of vectors*

Description

`vec_type()` finds the prototype of a single vector. `vec_type_common()` finds the common type of multiple vectors. `vec_ptype()` nicely prints the common type of any number of inputs, and is designed for interative exploration.

Usage

```
vec_type(x)

vec_type_common(..., .ptype = NULL)

vec_ptype(...)
```

Arguments

`..., x`	Vectors inputs
`.ptype`	If `NULL`, the default, the output type is determined by computing the common type across all elements of `...`. Alternatively, you can supply `.ptype` to give the output known type. If `getOption("vctrs.no_guessing` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.

Details

`vec_type_common()` first finds the prototype of each input, then finds the common type using `vec_type2()` and `Reduce()`.

Value

`vec_type()` and `vec_type_common()` return a prototype (a size-0 vector)

Prototype

A prototype is size 0 vector containing attributes, but no data. Generally, this is just `vec_slice(x, 0L)`, but some inputs require special handling.

For example, the prototype of logical vectors that only contain missing values is the special unspecified type, which can be coerced to any other 1d type. This allows bare `NA`s to represent missing values for any 1d vector type.

Examples

```
# Unknown types -----------------------------------------
vec_ptype()
vec_ptype(NA)
vec_ptype(NULL)

# Vectors -----------------------------------------------
vec_ptype(1:10)
vec_ptype(letters)
vec_ptype(TRUE)

vec_ptype(Sys.Date())
vec_ptype(Sys.time())
vec_ptype(factor("a"))
vec_ptype(ordered("a"))

# Matrices ----------------------------------------------
# The prototype of a matrix includes the number of columns
vec_ptype(array(1, dim = c(1, 2)))
vec_ptype(array("x", dim = c(1, 2)))

# Data frames -------------------------------------------
# The prototype of a data frame includes the prototype of
# every column
vec_ptype(iris)

# The prototype of multiple data frames includes the prototype
# of every column that in any data frame
vec_ptype(
  data.frame(x = TRUE),
  data.frame(y = 2),
  data.frame(z = "a")
)
```

`vec_unique` *Find and count unique values*

Description

- `vec_unique()`: the unique values. Equivalent to `unique()`.
- `vec_unique_loc()`: the locations of the unique values.
- `vec_unique_count()`: the number of unique values.

Usage

```
vec_unique(x)

vec_unique_loc(x)

vec_unique_count(x)
```

Arguments

x	A vector (including a data frame).

Value

- `vec_unique()`: a vector the same type as `x` containining only unique values.
- `vec_unique_loc()`: an integer vector, giving locations of unique values.
- `vec_unique_count()`: an integer vector of length 1, giving the number of unique values.

Missing values

In most cases, missing values are not considered to be equal, i.e. `NA == NA` is not `TRUE`. This behaviour would be unappealing here, so these functions consider all `NA`s to be equal. (Similarly, all `NaN` are also considered to be equal.)

Performance

These functions are currently slightly slower than their base equivalents. This is primarily because they do a little more checking and coercion in R, which makes them both a litter safer and more generic. Additionally, the C code underlying vctrs has not yet been implemented: we expect some performance improvements when that happens.

See Also

vec_duplicate for functions that work with the dual of unique values: duplicated values.

Examples

```
x <- rpois(100, 8)
vec_unique(x)
vec_unique_loc(x)
vec_unique_count(x)

# `vec_unique()` returns values in the order that encounters them
# use sort = "location" to match to the result of `vec_count()`
head(vec_unique(x))
head(vec_count(x, sort = "location"))

# Normally missing values are not considered to be equal
NA == NA

# But they are for the purposes of considering uniqueness
```

```
vec_unique(c(NA, NA, NA, NA, 1, 2, 1))
```

`%0%` *Default value for empty vectors*

Description

Use this inline operator when you need to provide a default value for empty (as defined by `vec_empty()`) vectors.

Usage

```
x %0% y
```

Arguments

`x`	A vector
`y`	Value to use to `x` is empty. To preserve type-stability, should be the same type as `x`.

Examples

```
1:10 %0% 5
integer() %0% 5
```

Index