Package 'vctrs'

July 2, 2019

Title Vector Helpers

Version 0.2.0

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 (>= 3.2)

Imports backports,
ellipsis (>= 0.2.0),
digest,
glue,
rlang (>= 0.4.0),
zeallot

Suggests bit64,
covr,
crayon,
generics,
knitr,
pillar (>= 1.4.1),
pkgdown,
rmarkdown,
testthat,
tibble

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 6.1.1

R topics documented:

---

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, ...)

validate_list_of(x)

is_list_of(x)

## S3 method for class 'vctrs_list_of'
vec_ptype2(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_guess` 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_ptype2()` and `vec_cast()`.

Details

Unlike regular lists, setting a list element to `NULL` using `[[` does not remove it.

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))
```

---

name_spec	*Name specifications*

Description

A name specification describes how to combine an inner and outer names. This sort of name combination arises when concatenating vectors or flattening lists. There are two possible cases:

- Named vector:

  ```
  vec_c(outer = c(inner1 = 1, inner2 = 2))
  ```

- Unnamed vector:

  ```
  vec_c(outer = 1:2)
  ```

In r-lib and tidyverse packages, these cases are errors by default, because there's no behaviour that works well for every case. Instead, you can provide a name specification that describes how to combine the inner and outer names of inputs. Name specifications can refer to:

- `outer`: The external name recycled to the size of the input vector.

- `inner`: Either the names of the input vector, or a sequence of integer from 1 to the size of the vector if it is unnamed.

Arguments

name_spec, .name_spec

A name specification for combining inner and outer names. This is relevant for inputs passed with a name, when these inputs are themselves named, like outer = c(inner = 1), or when they have length greater than 1: outer = 1:2. By default, these cases trigger an error. You can resolve the error by providing a specification that describes how to combine the names or the indices of the inner vector with the name of the input. This specification can be:

- A function of two arguments. The outer name is passed as a string to the first argument, and the inner names or positions are passed as second argument.
- An anonymous function as a purrr-style formula.
- A glue specification of the form "{outer}_{inner}".

See the name specification topic.

Examples

```
# By default, named inputs must be length 1:
vec_c(name = 1)         # ok
try(vec_c(name = 1:3))  # bad

# They also can't have internal names, even if scalar:
try(vec_c(name = c(internal = 1)))  # bad

# Pass a name specification to work around this. A specification
# can be a glue string referring to `outer` and `inner`:
vec_c(name = 1:3, other = 4:5, .name_spec = "{outer}")
vec_c(name = 1:3, other = 4:5, .name_spec = "{outer}_{inner}")

# They can also be functions:
my_spec <- function(outer, inner) paste(outer, inner, sep = "_")
vec_c(name = 1:3, other = 4:5, .name_spec = my_spec)

# Or purrr-style formulas for anonymous functions:
vec_c(name = 1:3, other = 4:5, .name_spec = ~ paste0(.x, .y))
```

---

partial_factor   *Partially specify a factor*

Description

This special class can be passed as a ptype in order to specify that the result should be a factor that contains at least the specified levels.

Usage

```
partial_factor(levels = character())
```

Arguments

levels   Character vector of labels.

Examples

```
# Assert that `x` is a factor
vec_assert(factor("x"), partial_factor())

# Testing with `factor()` is too strict,
# because it tries to match the levels exactly
# rather than learning them from the data.
try(vec_assert(factor("x"), factor()))

# You can also enforce a minimum set of levels
try(vec_assert(factor("x"), partial_factor("y")))

vec_assert(factor(c("x", "y")), partial_factor("y"))

pf <- partial_factor(levels = c("x", "y"))
pf

vec_ptype_common(factor("v"), factor("w"), .ptype = pf)
```

| 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

- `vec_is()` is a predicate that checks if its input conforms to a prototype and/or a size.

- `vec_assert()` throws an error when the input doesn't conform.

Usage

```
vec_assert(x, ptype = NULL, size = NULL,
  arg = as_label(substitute(x)))

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

Arguments

x	A vector argument to check.
ptype	Prototype to compare against. If the prototype has a class, its vec_ptype() is compared to that of x with `identical()`. Otherwise, its typeof() is compared to that of x with ==.
size	Size to compare against
arg	Name of argument being checked. This is used in error messages. The label of the expression passed as x is taken as default.

Value

`vec_is()` returns `TRUE` or `FALSE`. `vec_assert()` either throws a typed error (see section on error types) or returns x, invisibly.

Error types

- If the prototype doesn't match, an error of class `"vctrs_error_assert_ptype"` is raised.

- If the size doesn't match, an error of class `"vctrs_error_assert_size"` is raised.

Both errors inherit from `"vctrs_error_assert"`.

| vec_as_names | *Retrieve and repair names* |

Description

`vec_as_names()` takes a character vector of names and repairs it according to the `repair` argument. It is the r-lib and tidyverse equivalent of base::make.names().

vctrs deals with a few levels of name repair:

- `minimal` names exist. The `names` attribute is not `NULL`. The name of an unnamed element is `""` and never `NA`. For instance, `vec_as_names()` always returns minimal names and data frames created by the tibble package have names that are, at least, `minimal`.

- unique names are minimal, have no duplicates, and can be used where a variable name is expected. Empty names, and ... or .. followed by a sequence of digits are banned.
  - All columns can be accessed by name via df[["name"]] and df$`name` and with(df, `name`).
- universal names are unique and syntactic (see Details for more).
  - Names work everywhere, without quoting: df$name and with(df,name) and lm(name1 ~ name2,data = df) and dplyr::select(df,name) all work.

universal implies unique, unique implies minimal. These levels are nested.

Usage

```
vec_as_names(names, ..., repair = c("minimal", "unique", "universal",
  "check_unique"), quiet = FALSE)
```

Arguments

names	A character vector.
...	These dots are for future extensions and must be empty.
repair	Either a string or a function. If a string, it must be one of "check_unique", "minimal", "unique", or "universal". If a function, it is invoked with a vector of minimal names and must return minimal names, otherwise an error is thrown.

- Minimal names are never NULL or NA. When an element doesn't have a name, its minimal name is an empty string.
- Unique names are unique. A suffix is appended to duplicate names to make them unique.
- Universal names are unique and syntactic, meaning that you can safely use the names as variables without causing a syntax error.

The "check_unique" option doesn't perform any name repair. Instead, an error is raised if the names don't suit the "unique" criteria.

quiet	By default, the user is informed of any renaming caused by repairing the names. This only concerns unique and universal repairing. Set quiet to TRUE to silence the messages.

minimal **names**

minimal names exist. The names attribute is not NULL. The name of an unnamed element is "" and never NA.

Examples:

```
Original names of a vector with length 3: NULL
                        minimal names: "" "" ""

                      Original names: "x" NA
                       minimal names: "x" ""
```

unique **names**

unique names are minimal, have no duplicates, and can be used (possibly with backticks) in contexts where a variable is expected. Empty names, and ... or .. followed by a sequence of digits are banned If a data frame has unique names, you can index it by name, and also access the columns by name. In particular, df[["name"]] and df$`name` and also with(df, `name`) always work.

There are many ways to make names unique. We append a suffix of the form ...j to any name that is "" or a duplicate, where j is the position. We also change ..# and ... to ...#.

Example:

```
Original names:     ""      "x"     "" "y"      "x"  "..2"  "..."
  unique names: "...1" "x...2" "...3" "y" "x...5" "...6" "...7"
```

Pre-existing suffixes of the form ...j are always stripped, prior to making names unique, i.e. reconstructing the suffixes. If this interacts poorly with your names, you should take control of name repair.

universal **names**

universal names are unique and syntactic, meaning they:

- Are never empty (inherited from unique).

- Have no duplicates (inherited from unique).

- Are not .... Do not have the form ..i, where i is a number (inherited from unique).

- Consist of letters, numbers, and the dot . or underscore _ characters.

- Start with a letter or start with the dot . not followed by a number.

- Are not a [reserved](#) word, e.g., if or function or TRUE.

If a vector has universal names, variable names can be used "as is" in code. They work well with nonstandard evaluation, e.g., df$name works.

vctrs has a different method of making names syntactic than [base::make.names()](#). In general, vctrs prepends one or more dots . until the name is syntactic.

Examples:

```
 Original names:     ""      "x"     NA      "x"
universal names: "...1" "x...2" "...3" "x...4"

  Original names: "(y)"  "_z"  ".2fa"  "FALSE"
 universal names: ".y." "._z" "..2fa" ".FALSE"
```

See Also

[rlang::names2()](#) returns the names of an object, after making them minimal.

The [Names attribute](#) section in the "tidyverse package development principles".

Examples

```
# By default, `vec_as_names()` returns minimal names:
vec_as_names(c(NA, NA, "foo"))

# You can make them unique:
vec_as_names(c(NA, NA, "foo"), repair = "unique")

# Universal repairing fixes any non-syntactic name:
vec_as_names(c("_foo", "+"), repair = "universal")
```

---

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, .names_to = NULL,
  .name_repair = c("unique", "universal", "check_unique"))

vec_cbind(..., .ptype = NULL, .size = NULL,
  .name_repair = c("unique", "universal", "check_unique", "minimal"))
```

Arguments

`...`	Data frames or vectors.
	`vec_rbind()` ignores names unless `.names_to` is supplied. `vec_cbind()` creates packed data frame columns with named inputs.
	`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_guess` is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.
`.names_to`	Optionally, the name of a column where the names of `...` arguments are copied. These names are useful to identify which row comes from which input. If supplied, `...` must be named.
`.name_repair`	One of `"unique"`, `"universal"`, or `"check_unique"`. See [`vec_as_names()`](vec_as_names()) for the meaning of these options.
	With `vec_rbind()`, the repair function is applied to all inputs separately. This is because `vec_rbind()` needs to align their columns before binding the rows, and thus needs all inputs to have unique names. On the other hand, `vec_cbind()` applies the repair function after all inputs have been concatenated together in a final data frame. Hence `vec_cbind()` allows the more permissive minimal names repair.
`.size`	If, `NULL`, the default, will determine 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_ptype2() 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

All inputs are first converted to a data frame. The conversion for 1d vectors depends on the direction of binding:

- For vec_rbind(), each element of the vector becomes a column in a single row.
- For vec_cbind(), each element of the vector becomes a row in a single column.

Once the inputs have all become data frames, the following invariants are observed for row-binding:

- vec_size(vec_rbind(x,y)) == vec_size(x) + vec_size(y)
- vec_ptype(vec_rbind(x,y)) = vec_ptype_common(x,y)

Note that if an input is an empty vector, it is first converted to a 1-row data frame with 0 columns. Despite being empty, its effective size for the total number of rows is 1.

For column-binding, the following invariants apply:

- vec_size(vec_cbind(x,y)) == vec_size_common(x,y)
- vec_ptype(vec_cbind(x,y)) == vec_cbind(vec_ptype(x),vec_ptype(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]
)

# if you supply a named data frame, it is packed in a single column
data <- vec_cbind(
  x = data.frame(a = 1, b = 2),
  y = 1
)
data

# Packed data frames are nested in a single column. This makes it
# possible to access it through a single name:
data$x

# since the base print method is suboptimal with packed data
# frames, it is recommended to use tibble to work with these:
if (rlang::is_installed("tibble")) {
  vec_cbind(x = tibble::tibble(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, .name_spec = NULL,
   .name_repair = c("minimal", "unique", "check_unique", "universal"))
```

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_guess`is `TRUE` you must supply this value: this is a convenient way to make production code demand fixed types.
`.name_spec`	A name specification for combining inner and outer names. This is relevant for inputs passed with a name, when these inputs are themselves named, like `outer = c(inner = 1)`, or when they have length greater than 1: `outer = 1:2`. By default, these cases trigger an error. You can resolve the error by providing a specification that describes how to combine the names or the indices of the inner vector with the name of the input. This specification can be:

- A function of two arguments. The outer name is passed as a string to the first argument, and the inner names or positions are passed as second argument.
- An anonymous function as a purrr-style formula.
- A glue specification of the form `"{outer}_{inner}"`.

See the [name specification topic.](#)

`.name_repair`	How to repair names, see `repair` options in [`vec_as_names()`.](#)

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_ptype(vec_c(x,y)) == vec_ptype_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"))


# By default, named inputs must be length 1:
vec_c(name = 1)
try(vec_c(name = 1:3))

# Pass a name specification to work around this:
vec_c(name = 1:3, .name_spec = "{outer}_{inner}")

# See `?name_spec` for more examples of name specifications.
```

---

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 (named) atomic vector or list.

- `vec_data()` returns unstructured data. The only attributes preserved are names, dims, and dimnames.

  Currently, due to the underlying memory architecture of R, this creates a full copy of the data.

- `vec_proxy()` may return structured data. This generic is the main customisation point in vctrs, along with [vec_restore()]. See the section below to learn when you should implement `vec_proxy()`.

  Methods must return a vector type. Records and data frames will be processed rowwise.

Usage

```
vec_data(x)

vec_proxy(x, ...)
```

Arguments

x	A vector or object implementing `vec_proxy()`.
...	These dots are for future extensions and must be empty.

Value

The data underlying x, free from any attributes except the names.

When should you proxy your type

You should only implement `vec_proxy()` when your type is designed around a non-vector class. I.e. anything that is not either:

- An atomic vector
- A bare list
- A data frame

In this case, implement `vec_proxy()` to return such a vector class. The vctrs operations such as [vec_slice()] are applied on the proxy and `vec_restore()` is called to restore the original representation of your type.

The most common case where you need to implement `vec_proxy()` is for S3 lists. In vctrs, S3 lists are treated as scalars by default. This way we don't treat objects like model fits as vectors. To prevent vctrs from treating your S3 list as a scalar, unclass it in the `vec_proxy()` method. For instance, here is the definition for `list_of`:

```
vec_proxy.vctrs_list_of <- function(x) {
  unclass(x)
}
```

Another case where you need to implement a proxy is record types. Record types should return a data frame, as in the POSIXlt method:

```
vec_proxy.POSIXlt <- function(x) {
  new_data_frame(unclass(x))
}
```

Note that you don't need to implement vec_proxy() when your class inherits from vctrs_vctr or vctrs_rcrd.

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_default_cast   *Default cast method*

Description

This function should typically be called from the default vec_cast() method for your class, e.g. vec_cast.myclass.default(). It does two things:

- If x is an unspecified vector, it automatically casts it to to using vec_init().

- Otherwise, an error is thrown with stop_incompatible_cast().

Usage

```
vec_default_cast(x, to, x_arg = "x", to_arg = "to")
```

Arguments

x	Vectors to cast.
to	Type to cast to. If NULL, x will be returned as is.
x_arg	Argument names for x and to. These are used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).
to_arg	Argument names for x and to. These are used in error messages to inform the user about the locations of incompatible types (see stop_incompatible_type()).

---

vec_duplicate *Find duplicated values*

Description

- `vec_duplicate_any()`: detects the presence of duplicated values, similar to `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 giving 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.)

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

vec_equal_na() tests a special case: equality with NA. It is similar to is.na but:

- Considers the missing element of a list to be NULL.

- Considered data frames and records to be missing if every component is missing. This preserves the invariant that vec_equal_na(x) is equal to vec_equal(x,vec_init(x),na_equal = TRUE).

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 the same size as. 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, NA), y = c(1, 2, 1, NA, NA))
vec_equal(df, data.frame(x = 1, y = 2))
vec_equal_na(df)
```

---

vec_init *Initialize a vector*

Description

Initialize a vector

Usage

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

Arguments

x	Template of vector to initialize.
n	Desired size of result.

Examples

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

---

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 be-
haviour would be unappealing here, so these functions consider all `NA`s to be equal. (Similarly, all
`NaN` are also considered to be equal.)

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_order   *Order and sort vectors*

Description

Order and sort vectors

Usage

```
vec_order(x, direction = c("asc", "desc"), na_value = c("largest",
  "smallest"))

vec_sort(x, direction = c("asc", "desc"), na_value = c("largest",
  "smallest"))
```

Arguments

x	A vector
direction	Direction to sort in. Defaults to ascending.
na_value	Should `NA`s be treated as the largest or smallest values?

Value

- `vec_order()` an integer vector the same size as x.
- `vec_sort()` a vector with the same size and type as x.

Examples

```
x <- round(c(runif(9), NA), 3)
vec_order(x)
vec_sort(x)
vec_sort(x, "desc")

# Can also handle data frames
df <- data.frame(g = sample(2, 10, replace = TRUE), x = x)
```

```
vec_order(df)
vec_sort(df)
vec_sort(df, "desc")
```

---

vec_ptype   *Find the prototype of a set of vectors*

Description

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

Usage

```
vec_ptype(x)

vec_ptype_common(..., .ptype = NULL)

vec_ptype_show(...)
```

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_guess is TRUE you must supply this value: this is a convenient way to make production code demand fixed types.

Details

vec_ptype_common() first finds the prototype of each input, then successively calls [vec_ptype2()](#) to find a common type.

Value

vec_ptype() and vec_ptype_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 NAs to represent missing values for any 1d vector type.

The prototype of NULL

In general we treat NULL as an absence of argument (which is often an error). In the case of vec_ptype() and vec_ptype_common(), we have chosen NULL as the identity of the common type monoid: the common type of foo and NULL is always foo. For this reason, the prototype of NULL is NULL.

Examples

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

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

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

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

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

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

---

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 NULLs 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)
vec_recycle_common(integer(), 5)
## Not run:
vec_recycle_common(1:5, 1:2)

## End(Not run)

# 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_repeat   *Expand the length of a vector*

Description

This is a special case of [rep()](#) for the special case of integer `times` and `each` values, and works along size, rather than length.

Usage

```
vec_repeat(x, each = 1L, times = 1L)
```

Arguments

x	A vector.
each	Number of times to repeat each element of x.
times	Number of times to repeat the whole vector of x.

Value

A vector the same type as x with size `vec_size(x) * times * each`.

Examples

```
# each repeats within
vec_repeat(1:3, each = 2)
# times repeats whole thing
vec_repeat(1:3, times = 2)

df <- data.frame(x = 1:2, y = 1:2)
# rep() repeats columns of data frame, and returns list:
rep(df, each = 2)
# vec_repeat() repeats rows, and returns same data.frame
vec_repeat(df, 2)
```

---

vec_seq_along   *Useful sequences*

Description

vec_seq_along() is equivalent to [seq_along()](#) but uses size, not length. vec_init_along() creates a vector of missing values with size matching an existing object.

Usage

```
vec_seq_along(x)

vec_init_along(x, y = x)
```

Arguments

x, y	Vectors

Value

- `vec_seq_along()` an integer vector with the same size as `x`.
- `vec_init_along()` a vector with the same type as `x` and the same size as `y`.

Examples

```
vec_seq_along(mtcars)
vec_init_along(head(mtcars))
```

---

vec_size	*Number of observations*

Description

`vec_size(x)` returns the size of a vector. `vec_is_empty()` returns `TRUE` if the size is zero, `FALSE` otherwise.

The size 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, .absent = 0L)

vec_is_empty(x)
```

Arguments

x, ...	Vector inputs or `NULL`.
.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.
.absent	The size used when no input is provided, or when all input is `NULL`. If left as `NULL` when no input is supplied, an error is thrown.

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).

`vec_size_common()` returns `.absent` if all inputs are `NULL` or absent, `0L` by default.

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)`

The size of NULL

The size of NULL is hard-coded to `0L` in `vec_size()`. `vec_size_common()` returns `.absent` when all inputs are NULL (if only some inputs are NULL, they are simply ignored).

A default size of 0 makes sense because sizes are most often queried in order to compute a total size while assembling a collection of vectors. Since we treat NULL as an absent input by principle, we return the identity of sizes under addition to reflect that an absent input doesn't take up any size.

Note that other defaults might make sense under different circumstances. For instance, a default size of 1 makes sense for finding the common size because 1 is the identity of the recycling rules.

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

---

vec_split	*Split a vector into groups*

Description

This is a generalisation of [split()](#) that can split by any type of vector, not just factors. Instead of returning the keys in the character names, the are returned in a separate parallel vector.

Usage

```
vec_split(x, by)
```

Arguments

x	Vector to divide into groups.
by	Vector whose unique values defines the groups.

Value

A data frame with two columns and size equal to vec_size(vec_unique(by)). The key column
has the same type as by, and the val column has type list_of<vec_ptype(x)>.

Note for complex types, the default data.frame print method will be suboptimal, and you will
want to coerce into a tibble to better understand the output.

Examples

```
vec_split(mtcars$cyl, mtcars$vs)
vec_split(mtcars$cyl, mtcars[c("vs", "am")])

if (require("tibble")) {
  as_tibble(vec_split(mtcars$cyl, mtcars[c("vs", "am")]))
  as_tibble(vec_split(mtcars, mtcars[c("vs", "am")]))
}
```

---

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 be-
haviour would be unappealing here, so these functions consider all NAs to be equal. (Similarly, all
NaN are also considered to be equal.)

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_is_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