Package 'purrr'

May 11, 2017

Title Functional Programming Tools

Version 0.2.2.2

Description Make your pure functions purr with the 'purrr' package. This package completes R's functional programming tools with missing features present in other programming languages.

License GPL-3 | file LICENSE

LazyData true

Imports magrittr (>= 1.5), Rcpp, lazyeval (>= 0.2.0), tibble

Suggests testthat, covr, dplyr (>= 0.4.3)

URL https://github.com/hadley/purrr

BugReports https://github.com/hadley/purrr/issues

RoxygenNote 6.0.1

NeedsCompilation yes

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

Maintainer Lionel Henry <lionel@rstudio.com>

Repository CRAN

Date/Publication 2017-05-11 18:22:22 UTC

R topics documented:

`accumulate` *Accumulate recursive folds across a list*

Description

`accumulate` applies a function recursively over a list from the left, while `accumulate_right` applies the function from the right. Unlike `reduce` both functions keep the intermediate results.

Usage

```
accumulate(.x, .f, ..., .init)

accumulate_right(.x, .f, ..., .init)
```

Arguments

`.x`	A list or atomic vector.
`.f`	A two-argument function.
`...`	Additional arguments passed on to `.f`.
`.init`	If supplied, will be used as the first value to start the accumulation, rather than using `x[[1]]`. This is useful if you want to ensure that `reduce` returns the correct value when `.x` is `is_empty()`.

Examples

```
1:3 %>% accumulate(`+`)
1:10 %>% accumulate_right(`*`)

# From Haskell's scanl documentation
1:10 %>% accumulate(max, .init = 5)

# Simulating stochastic processes with drift
## Not run:
library(dplyr)
library(ggplot2)

rerun(5, rnorm(100)) %>%
  set_names(paste0("sim", 1:5)) %>%
  map(~ accumulate(., ~ .05 + .x + .y)) %>%
  map_df(~ data_frame(value = .x, step = 1:100), .id = "simulation") %>%
  ggplot(aes(x = step, y = value)) +
    geom_line(aes(color = simulation)) +
    ggtitle("Simulations of a random walk with drift")

## End(Not run)
```

---

along *Helper to create vectors with matching length.*

Description

These functions take the idea of `seq_along` and generalise it to creating lists (`list_along`) and repeating values (`rep_along`).

Usage

```
list_along(x)

rep_along(x, y)
```

Arguments

x	A vector.
y	Values to repeat.

Examples

```
x <- 1:5
rep_along(x, 1:2)
rep_along(x, 1)
list_along(x)
```

---

array-coercion *Coerce array to list*

Description

`array_branch()` and `array_tree()` enable arrays to be used with purrr's functionals by turning them into lists. The details of the coercion are controlled by the `margin` argument. `array_tree()` creates an hierarchical list (a tree) that has as many levels as dimensions specified in `margin`, while `array_branch()` creates a flat list (by analogy, a branch) along all mentioned dimensions.

Usage

```
array_branch(array, margin = NULL)

array_tree(array, margin = NULL)
```

Arguments

array	An array to coerce into a list.
margin	A numeric vector indicating the positions of the indices to be to be enlisted. If `NULL`, a full margin is used. If `numeric(0)`, the array as a whole is wrapped in a list.

Details

When no margin is specified, all dimensions are used by default. When `margin` is a numeric vector of length zero, the whole array is wrapped in a list.

Examples

```
# We create an array with 3 dimensions
x <- array(1:12, c(2, 2, 3))

# A full margin for such an array would be the vector 1:3. This is
# the default if you don't specify a margin

# Creating a branch along the full margin is equivalent to
# as.list(array) and produces a list of size length(x):
array_branch(x) %>% str()

# A branch along the first dimension yields a list of length 2
# with each element containing a 2x3 array:
array_branch(x, 1) %>% str()

# A branch along the first and third dimensions yields a list of
# length 2x3 whose elements contain a vector of length 2:
array_branch(x, c(1, 3)) %>% str()

# Creating a tree from the full margin creates a list of lists of
# lists:
array_tree(x) %>% str()

# The ordering and the depth of the tree are controlled by the
# margin argument:
array_tree(x, c(3, 1)) %>% str()
```

---

as_function *Convert an object into a function.*

Description

as_function is the powerhouse behind the varied function specifications that purrr functions allow. This is an S3 generic so that other people can make as_function work with their own objects.

Usage

```
as_function(.f, ...)

## S3 method for class 'character'
as_function(.f, ..., .null = NULL)
```

Arguments

.f	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. ~ .x + 2, it is converted to a function with two arguments, .x or . and .y. This allows you to create very compact anonymous functions with up to two inputs.

	If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to methods.
`.null`	Optional additional argument for character and numeric inputs.

Examples

```
as_function(~ . + 1)
as_function(1)
as_function(c("a", "b", "c"))
as_function(c("a", "b", "c"), .null = NA)
```

`as_vector` *Coerce a list to a vector*

Description

`as_vector()` collapses a list of vectors into one vector. It checks that the type of each vector is consistent with `.type`. If the list can not be simplified, it throws an error. `simplify` will simplify a vector if possible; `simplify_all` will apply `simplify` to every element of a list.

Usage

```
as_vector(.x, .type = NULL)

simplify(.x, .type = NULL)

simplify_all(.x, .type = NULL)
```

Arguments

`.x`	A list of vectors
`.type`	A vector mold or a string describing the type of the input vectors. The latter can be any of the types returned by `typeof()`, or "numeric" as a shorthand for either "double" or "integer".

Details

`.type` can be a vector mold specifying both the type and the length of the vectors to be concatenated, such as `numeric(1)` or `integer(4)`. Alternatively, it can be a string describing the type, one of: "logical", "integer", "double", "complex", "character" or "raw".

Examples

```
# Supply the type either with a string:
as.list(letters) %>% as_vector("character")

# Or with a vector mold:
as.list(letters) %>% as_vector(character(1))

# Vector molds are more flexible because they also specify the
# length of the concatenated vectors:
list(1:2, 3:4, 5:6) %>% as_vector(integer(2))

# Note that unlike vapply(), as_vector() never adds dimension
# attributes. So when you specify a vector mold of size > 1, you
# always get a vector and not a matrix
```

---

at_depth *Map a function over lower levels of a nested list*

Description

`at_depth()` maps a function on lower levels of nested lists. In essence, `at_depth()` is a recursive map.

Usage

```
at_depth(.x, .depth, .f, ...)
```

Arguments

`.x`	A deep list
`.depth`	Level of `.x` to map on.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to `.f`.

Details

- `x %>% at_depth(0, fun)` is equivalent to `fun(x)`.
- `x %>% at_depth(1, fun)` is equivalent to `map(x, fun)`.
- `x %>% at_depth(2, fun)` is equivalent to `map(x, . %>% map(fun))`.

Examples

```
l1 <- list(
  obj1 = list(
    prop1 = list(param1 = 1:2, param2 = 3:4),
    prop2 = list(param1 = 5:6, param2 = 7:8)
  ),
  obj2 = list(
    prop1 = list(param1 = 9:10, param2 = 11:12),
    prop2 = list(param1 = 13:14, param2 = 15:16)
  )
)

# In the above list, "obj" is level 1, "prop" is level 2 and "param"
# is level 3. To apply sum() on all params, we map it at depth 3:
l1 %>% at_depth(3, sum)

# map() lets us pluck the elements prop1/param2 in obj1 and obj2:
l1 %>% map(c("prop1", "param2")) %>% str()

# But what if we want to pluck all param2 elements? Then we need to
# act at a lower level:
l1 %>% at_depth(2, "param2") %>% str()


# at_depth can be used in a complementary way with other purrr
# functions to make them operate at a lower level
l2 <- list(
  obj1 = list(
    prop1 = list(c(1, 2), c(3, 4), c(5, 6)),
    prop2 = list(c("a", "b"), c("c", "d"), c("e", "f"))
  ),
  obj2 = list(
    prop1 = list(c(10, 20), c(30, 40), c(50, 60)),
    prop2 = list(c("A", "B"), c("C", "D"), c("E", "F"))
  )
)

# Here we ask pmap() to map paste() simultaneously over all
# elements of the objects at the second level. paste() is thus
# effectively mapped at level 3.
l2 %>% at_depth(2, pmap, paste, sep = " / ")
```

---

bare-type-predicates *Bare type predicates*

Description

These predicates check for a given type but only return `TRUE` for bare R objects. Bare objects have no class attributes. For example, a data frame is a list, but not a bare list.

Usage

```
is_bare_list(x)

is_bare_atomic(x)

is_bare_vector(x)

is_bare_double(x)

is_bare_integer(x)

is_bare_numeric(x)

is_bare_character(x)

is_bare_logical(x)
```

Arguments

x	object to be tested.

Details

- Like `is_atomic()` and unlike base R `is.atomic()`, `is_bare_atomic()` does not return `TRUE` for `NULL`.
- Unlike base R `is.numeric()`, `is_bare_double()` only returns `TRUE` for floating point numbers.

See Also

type-predicates scalar-type-predicates

---

compose *Compose multiple functions*

Description

Compose multiple functions

Usage

```
compose(...)
```

Arguments

...	n functions to apply in order from right to left.

Examples

```
not_null <- compose(`!`, is.null)
not_null(4)
not_null(NULL)

add1 <- function(x) x + 1
compose(add1, add1)(8)
```

`conditional-map` *Modify elements conditionally*

Description

`map_if()` maps a function over the elements of `.x` satisfying a predicate. `map_at()` is similar but will modify the elements corresponding to a character vector of names or a mumeric vector of positions.

Usage

```
map_if(.x, .p, .f, ...)

map_at(.x, .at, .f, ...)
```

Arguments

`.x`	A list or atomic vector.
`.p`	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as `.x`. Alternatively, if the elements of `.x` are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where `.p` evaluates to `TRUE` will be modified.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to `.f`.
`.at`	A character vector of names or a numeric vector of positions. Only those elements corresponding to `.at` will be modified.

Value

A list.

Examples

```
# Convert factors to characters
iris %>%
  map_if(is.factor, as.character) %>%
  str()

# Specify which columns to map with a numeric vector of positions:
mtcars %>% map_at(c(1, 4, 5), as.character) %>% str()

# Or with a vector of names:
mtcars %>% map_at(c("cyl", "am"), as.character) %>% str()

list(x = rbernoulli(100), y = 1:100) %>%
  transpose() %>%
  map_if("x", ~ update_list(., y = ~ y * 100)) %>%
  transpose() %>%
  simplify_all()
```

---

contains *Does a list contain an object?*

Description

Does a list contain an object?

Usage

```
contains(.x, .y)
```

Arguments

.x	A list or atomic vector.
.y	Object to test for

Examples

```
x <- list(1:10, 5, 9.9)
x %>% contains(1:10)
x %>% contains(3)
```

cross_n *Produce all combinations of list elements*

Description

`cross2()` returns the product set of the elements of `.x` and `.y`. `cross3()` takes an additional `.z` argument. `cross_n()` takes a list `.l` and returns the cartesian product of all its elements in a list, with one combination by element. `cross_d()` is like `cross_n()` but returns a data frame, with one combination by row.

Usage

```
cross_n(.l, .filter = NULL)

cross2(.x, .y, .filter = NULL)

cross3(.x, .y, .z, .filter = NULL)

cross_d(.l, .filter = NULL)
```

Arguments

`.l`	A list of lists or atomic vectors. Alternatively, a data frame. `cross_d()` requires all elements to be named.
`.filter`	A predicate function that takes the same number of arguments as the number of variables to be combined.
`.x, .y, .z`	Lists or atomic vectors.

Details

`cross_n()`, `cross2()` and `cross3()` return the cartesian product is returned in wide format. This makes it more amenable to mapping operations. `cross_d()` returns the output in long format just as `expand.grid()` does. This is adapted to rowwise operations.

When the number of combinations is large and the individual elements are heavy memory-wise, it is often useful to filter unwanted combinations on the fly with `.filter`. It must be a predicate function that takes the same number of arguments as the number of crossed objects (2 for `cross2()`, 3 for `cross3()`, `length(.l)` for `cross_n()`) and returns `TRUE` or `FALSE`. The combinations where the predicate function returns `TRUE` will be removed from the result.

Value

`cross2()`, `cross3()` and `cross_n()` always return a list. `cross_d()` always returns a data frame. `cross_n()` returns a list where each element is one combination so that the list can be directly mapped over. `cross_d()` returns a data frame where each row is one combination.

See Also

expand.grid()

Examples

```
# We build all combinations of names, greetings and separators from our
# list of data and pass each one to paste()
data <- list(
  id = c("John", "Jane"),
  greeting = c("Hello.", "Bonjour."),
  sep = c("! ", "... ")
)

data %>%
  cross_n() %>%
  map(lift(paste))

# cross_n() returns the combinations in long format: many elements,
# each representing one combination. With cross_d() we'll get a
# data frame in long format: crossing three objects produces a data
# frame of three columns with each row being a particular
# combination. This is the same format that expand.grid() returns.
args <- data %>% cross_d()

# In case you need a list in long format (and not a data frame)
# just run as.list() after cross_d()
args %>% as.list()

# This format is often less pratical for functional programming
# because applying a function to the combinations requires a loop
out <- vector("list", length = nrow(args))
for (i in seq_along(out))
  out[[i]] <- map(args, i) %>% invoke(paste, .)
out

# It's easier to transpose and then use invoke_map()
args %>% transpose() %>% map_chr(~ invoke(paste, .))

# Unwanted combinations can be filtered out with a predicate function
filter <- function(x, y) x >= y
cross2(1:5, 1:5, .filter = filter) %>% str()

# To give names to the components of the combinations, we map
# setNames() on the product:
seq_len(3) %>%
  cross2(., ., .filter = `==`) %>%
  map(setNames, c("x", "y"))

# Alternatively we can encapsulate the arguments in a named list
# before crossing to get named components:
seq_len(3) %>%
  list(x = ., y = .) %>%
```

```
cross_n(.filter = `==`)
```

---

detect *Find the value or position of the first match.*

Description

Find the value or position of the first match.

Usage

```
detect(.x, .p, ..., .right = FALSE)

detect_index(.x, .p, ..., .right = FALSE)
```

Arguments

.x	A list or atomic vector.
.p	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as .x. Alternatively, if the elements of .x are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where .p evaluates to TRUE will be modified.
...	Additional arguments passed on to .f.
.right	If FALSE, the default, starts at the beginnging of the vector and move towards the end; if TRUE, starts at the end of the vector and moves towards the beginning.

Value

detect the value of the first item that matches the predicate; detect_index the position of the matching item. If not found, detect returns NULL and detect_index returns 0.

Examples

```
is_even <- function(x) x %% 2 == 0

3:10 %>% detect(is_even)
3:10 %>% detect_index(is_even)

3:10 %>% detect(is_even, .right = TRUE)
3:10 %>% detect_index(is_even, .right = TRUE)
```

every — *Do every or some elements of a list satisfy a predicate?*

Description

Do every or some elements of a list satisfy a predicate?

Usage

```
every(.x, .p, ...)

some(.x, .p, ...)
```

Arguments

.x	A list or atomic vector.
.p	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as .x. Alternatively, if the elements of .x are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where .p evaluates to TRUE will be modified.
...	Additional arguments passed on to .f.

Value

Either TRUE or FALSE.

Examples

```
x <- list(0, 1, TRUE)
x %>% every(identity)
x %>% some(identity)

y <- list(0:10, 5.5)
y %>% every(is.numeric)
y %>% every(is.integer)
```

flatten — *Flatten a list of lists into a simple vector.*

Description

These functions remove a level hierarchy from a list. They are similar to unlist, only ever remove a single layer of hierarchy, and are type-stable so you always know what the type of the output is.

Usage

```
flatten(.x)

flatten_lgl(.x)

flatten_int(.x)

flatten_dbl(.x)

flatten_chr(.x)

flatten_df(.x, .id = NULL)
```

Arguments

.x	A list of flatten. The contents of the list can be anything for `flatten` (as a list is returned), but the contents must match the type for the other functions.
.id	If not `NULL` a variable with this name will be created giving either the name or the index of the data frame.

Value

`flatten()` returns a list, `flatten_lgl` a logical vector, `flatten_int` an integer vector, `flatten_dbl` a double vector, and `flatten_chr` a character vector.

Examples

```
x <- rerun(2, sample(4))
x
x %>% flatten()
x %>% flatten_int()

# You can use flatten in conjunction with map
x %>% map(1L) %>% flatten_int()
# But it's more efficient to use the typed map instead.
x %>% map_int(1L)
```

---

get-attr *Infix attribute accessor*

Description

Infix attribute accessor

Usage

```
x %@% name
```

Arguments

x	Object
name	Attribute name

Examples

```
factor(1:3) %@% "levels"
mtcars %@% "class"
```

---

head_while *Find head/tail that all satisfies a predicate.*

Description

Find head/tail that all satisfies a predicate.

Usage

```
head_while(.x, .p, ...)

tail_while(.x, .p, ...)
```

Arguments

.x	A list or atomic vector.
.p	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as .x. Alternatively, if the elements of .x are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where .p evaluates to TRUE will be modified.
...	Additional arguments passed on to .f.

Examples

```
pos <- function(x) x >= 0
head_while(5:-5, pos)
tail_while(5:-5, negate(pos))

big <- function(x) x > 100
head_while(0:10, big)
tail_while(0:10, big)
```

invoke *Invoke functions.*

Description

This pair of functions make it easier to combine a function and list of parameters to get a result. `invoke` is a wrapper around `do.call` that makes it easy to use in a pipe. `invoke_map` makes it easier to call lists of functions with lists of parameters.

Usage

```
invoke(.f, .x = NULL, ..., .env = NULL)

invoke_map(.f, .x = list(NULL), ..., .env = NULL)

invoke_map_lgl(.f, .x = list(NULL), ..., .env = NULL)

invoke_map_int(.f, .x = list(NULL), ..., .env = NULL)

invoke_map_dbl(.f, .x = list(NULL), ..., .env = NULL)

invoke_map_chr(.f, .x = list(NULL), ..., .env = NULL)

invoke_map_df(.f, .x = list(NULL), ..., .env = NULL)
```

Arguments

`.f`	For `invoke`, a function; for `invoke_map` a list of functions.
`.x`	For `invoke`, an argument-list; for `invoke_map` a list of argument-lists the same length as `.f` (or length 1). The default argument, `list(NULL)`, will be recycled to the same length as `.f`, and will call each function with no arguments (apart from any suppled in `...`.
`...`	Additional arguments passed to each function.
`.env`	Environment in which `do.call()` should evaluate a constructed expression. This only matters if you pass as `.f` the name of a function rather than its value, or as `.x` symbols of objects rather than their values.

Examples

```
# Invoke a function with a list of arguments
invoke(runif, list(n = 10))
# Invoke a function with named arguments
invoke(runif, n = 10)

# Combine the two:
invoke(paste, list("01a", "01b"), sep = "-")
# That's more natural as part of a pipeline:
```

```
list("01a", "01b") %>%
  invoke(paste, ., sep = ".")

# Invoke a list of functions, each with different arguments
invoke_map(list(runif, rnorm), list(list(n = 10), list(n = 5)))
# Or with the same inputs:
invoke_map(list(runif, rnorm), list(list(n = 5)))
invoke_map(list(runif, rnorm), n = 5)
# Or the same function with different inputs:
invoke_map("runif", list(list(n = 5), list(n = 10)))

# Or as a pipeline
list(m1 = mean, m2 = median) %>% invoke_map(x = rcauchy(100))
list(m1 = mean, m2 = median) %>% invoke_map_dbl(x = rcauchy(100))

# Note that you can also match by position by explicitly omitting `.x`.
# This can be useful when the argument names of the functions are not
# identical
list(m1 = mean, m2 = median) %>%
  invoke_map(, rcauchy(100))

# If you have pairs of function name and arguments, it's natural
# to store them in a data frame:
if (requireNamespace("dplyr", quietly = TRUE)) {
df <- dplyr::data_frame(
  f = c("runif", "rpois", "rnorm"),
  params = list(
    list(n = 10),
    list(n = 5, lambda = 10),
    list(n = 10, mean = -3, sd = 10)
  )
)
df
invoke_map(df$f, df$params)
}
```

`is_empty` *Is a vector/list empty?*

Description

Is a vector/list empty?

Usage

```
is_empty(x)
```

Arguments

x	object to test

Examples

```
is_empty(NULL)
is_empty(list())
is_empty(list(NULL))
```

`is_formula` *Is a formula?*

Description

Is a formula?

Usage

```
is_formula(x)
```

Arguments

`x`	object to test

Examples

```
x <- disp ~ am
is_formula(x)
```

`keep` *Keep or discard elements using a predicate function.*

Description

`keep` and `discard` are opposites. `compact` is a handy wrapper that removes all elements that are `NULL`.

Usage

```
keep(.x, .p, ...)

discard(.x, .p, ...)

compact(.x, .p = identity)
```

Arguments

`.x`	A list or vector.
`.p`	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as `.x`. Alternatively, if the elements of `.x` are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where `.p` evaluates to `TRUE` will be modified.
`...`	Additional arguments passed on to `.p`.

Details

These are usually called `select` or `filter` and `reject` or `drop`, but those names are already taken. `keep` is similar to `Filter` but the argument order is more convenient, and the evaluation of `.f` is stricter.

Examples

```
rep(10, 10) %>%
  map(sample, 5) %>%
  keep(function(x) mean(x) > 6)

# Or use a formula
rep(10, 10) %>%
  map(sample, 5) %>%
  keep(~ mean(.x) > 6)

# Using a string instead of a function will select all list elements
# where that subelement is TRUE
x <- rerun(5, a = rbernoulli(1), b = sample(10))
x
x %>% keep("a")
x %>% discard("a")
```

`lift` *Lift the domain of a function*

Description

`lift_xy()` is a composition helper. It helps you compose functions by lifting their domain from a kind of input to another kind. The domain can be changed from and to a list (l), a vector (v) and dots (d). For example, `lift_ld(fun)` transforms a function taking a list to a function taking dots.

Usage

```
lift(..f, ..., .unnamed = FALSE)

lift_dl(..f, ..., .unnamed = FALSE)
```

```
lift_dv(..f, ..., .unnamed = FALSE)

lift_vl(..f, ..., .type)

lift_vd(..f, ..., .type)

lift_ld(..f, ...)

lift_lv(..f, ...)
```

Arguments

`..f`	A function to lift.
`...`	Default arguments for `..f`. These will be evaluated only once, when the lifting factory is called.
`.unnamed`	If `TRUE`, `ld` or `lv` will not name the parameters in the lifted function signature. This prevents matching of arguments by name and match by position instead.
`.type`	A vector mold or a string describing the type of the input vectors. The latter can be any of the types returned by `typeof()`, or "numeric" as a shorthand for either "double" or "integer".

Details

The most important of those helpers is probably `lift_dl()` because it allows you to transform a regular function to one that takes a list. This is often essential for composition with purrr functional tools. Since this is such a common function, `lift()` is provided as an alias for that operation.

from ... to `list(...)` or `c(...)`

Here dots should be taken here in a figurative way. The lifted functions does not need to take dots per se. The function is simply wrapped a function in `do.call()`, so instead of taking multiple arguments, it takes a single named list or vector which will be interpreted as its arguments. This is particularly useful when you want to pass a row of a data frame or a list to a function and don't want to manually pull it apart in your function.

from `c(...)` to `list(...)` or `...`

These factories allow a function taking a vector to take a list or dots instead. The lifted function internally transforms its inputs back to an atomic vector. purrr does not obey the usual R casting rules (e.g., `c(1, "2")` produces a character vector) and will produce an error if the types are not compatible. Additionally, you can enforce a particular vector type by supplying `.type`.

from list(...) to c(...) or ...

`lift_ld()` turns a function that takes a list into a function that takes dots. `lift_vd()` does the same with a function that takes an atomic vector. These factory functions are the inverse operations of `lift_dl()` and `lift_dv()`.

`lift_vd()` internally coerces the inputs of `..f` to an atomic vector. The details of this coercion can be controlled with `.type`.

See Also

invoke()

Examples

```
### Lifting from ... to list(...) or c(...)

x <- list(x = c(1:100, NA, 1000), na.rm = TRUE, trim = 0.9)
lift_dl(mean)(x)

# Or in a pipe:
mean %>% lift_dl() %>% invoke(x)

# You can also use the lift() alias for this common operation:
lift(mean)(x)

# Default arguments can also be specified directly in lift_dl()
list(c(1:100, NA, 1000)) %>% lift_dl(mean, na.rm = TRUE)()

# lift_dl() and lift_ld() are inverse of each other.
# Here we transform sum() so that it takes a list
fun <- sum %>% lift_dl()
fun(list(3, NA, 4, na.rm = TRUE))

# Now we transform it back to a variadic function
fun2 <- fun %>% lift_ld()
fun2(3, NA, 4, na.rm = TRUE)

# It can sometimes be useful to make sure the lifted function's
# signature has no named parameters, as would be the case for a
# function taking only dots. The lifted function will take a list
# or vector but will not match its arguments to the names of the
# input. For instance, if you give a data frame as input to your
# lifted function, the names of the columns are probably not
# related to the function signature and should be discarded.
lifted_identical <- lift_dl(identical, .unnamed = TRUE)
mtcars[c(1, 1)] %>% lifted_identical()
mtcars[c(1, 2)] %>% lifted_identical()
#


### Lifting from c(...) to list(...) or ...

# In other situations we need the vector-valued function to take a
# variable number of arguments as with pmap(). This is a job for
# lift_vd():
pmap(mtcars, lift_vd(mean))

# lift_vd() will collect the arguments and concatenate them to a
# vector before passing them to ..f. You can add a check to assert
# the type of vector you expect:
lift_vd(tolower, .type = character(1))("this", "is", "ok")
```

```
#


### Lifting from list(...) to c(...) or ...

# cross_n() normally takes a list of elements and returns their
# cartesian product. By lifting it you can supply the arguments as
# if it was a function taking dots:
cross <- lift_ld(cross_n)
out1 <- cross_n(list(a = 1:2, b = c("a", "b", "c")))
out2 <- cross(a = 1:2, b = c("a", "b", "c"))
identical(out1, out2)

# This kind of lifting is sometimes needed for function
# composition. An example would be to use pmap() with a function
# that takes a list. In the following, we use some() on each row of
# a data frame to check they each contain at least one element
# satisfying a condition:
mtcars %>% pmap(lift_ld(some, partial(`<`, 200)))

# Default arguments for ..f can be specified in the call to
# lift_ld()
lift_ld(cross_n, .filter = `==`)(1:3, 1:3) %>% str()


# Here is another function taking a list and that we can update to
# take a vector:
glue <- function(l) {
  if (!is.list(l)) stop("not a list")
  l %>% invoke(paste, .)
}

## Not run:
letters %>% glue()           # fails because glue() expects a list
## End(Not run)

letters %>% lift_lv(glue)()  # succeeds
```

`lmap` *Apply a function to list-elements of a list*

Description

`lmap()`, `lmap_at()` and `lmap_if()` are similar to `map()`, `map_at()` and `map_if()`, with the difference that they operate exclusively on functions that take *and* return a list (or data frame). Thus, instead of mapping the elements of a list (as in `.x[[i]]`), they apply a function `.f` to each subset of size 1 of that list (as in `.x[i]`). We call those those elements 'list-elements').

Usage

```
lmap(.x, .f, ...)

lmap_if(.x, .p, .f, ...)

lmap_at(.x, .at, .f, ...)
```

Arguments

`.x`	A list or data frame.
`.f`	A function that takes and returns a list or data frame.
`...`	Additional arguments passed on to `.f`.
`.p`	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as `.x`. Alternatively, if the elements of `.x` are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where `.p` evaluates to `TRUE` will be modified.
`.at`	A character vector of names or a numeric vector of positions. Only those elements corresponding to `.at` will be modified.

Details

Mapping the list-elements `.x[i]` has several advantages. It makes it possible to work with functions that exclusively take a list or data frame. It enables `.f` to access the attributes of the encapsulating list, like the name of the components it receives. It also enables `.f` to return a larger list than the list-element of size 1 it got as input. Conversely, `.f` can also return empty lists. In these cases, the output list is reshaped with a different size than the input list `.x`.

Value

If `.x` is a list, a list. If `.x` is a data frame, a data frame.

See Also

`map_at()`, `map_if()` and `map()`

Examples

```
# Let's write a function that returns a larger list or an empty list
# depending on some condition. This function also uses the names
# metadata available in the attributes of the list-element
maybe_rep <- function(x) {
  n <- rpois(1, 2)
  out <- rep_len(x, n)
  if (length(out) > 0) {
    names(out) <- paste0(names(x), seq_len(n))
  }
  out
}
```

```
# The output size varies each time we map f()
x <- list(a = 1:4, b = letters[5:7], c = 8:9, d = letters[10])
x %>% lmap(maybe_rep)

# We can apply f() on a selected subset of x
x %>% lmap_at(c("a", "d"), maybe_rep)

# Or only where a condition is satisfied
x %>% lmap_if(is.character, maybe_rep)


# A more realistic example would be a function that takes discrete
# variables in a dataset and turns them into disjunctive tables, a
# form that is amenable to fitting some types of models.

# A disjunctive table contains only 0 and 1 but has as many columns
# as unique values in the original variable. Ideally, we want to
# combine the names of each level with the name of the discrete
# variable in order to identify them. Given these requirements, it
# makes sense to have a function that takes a data frame of size 1
# and returns a data frame of variable size.
disjoin <- function(x, sep = "_") {
  name <- names(x)
  x <- as.factor(x[[1]])

  out <- lapply(levels(x), function(level) {
    as.numeric(x == level)
  })

  names(out) <- paste(name, levels(x), sep = sep)
  dplyr::as_data_frame(out)
}

# Now, we are ready to map disjoin() on each categorical variable of a
# data frame:
iris %>% lmap_if(is.factor, disjoin)
mtcars %>% lmap_at(c("cyl", "vs", "am"), disjoin)
```

`map` *Apply a function to each element of a vector*

Description

`map()` returns the transformed input; `walk()` calls `.f` for its side-effect and returns the original input. `map()` returns a list or a data frame; `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()` return vectors of the corresponding type (or die trying); `map_df()` returns a data frame by row-binding the individual elements.

Usage

```
map(.x, .f, ...)

map_lgl(.x, .f, ...)

map_chr(.x, .f, ...)

map_int(.x, .f, ...)

map_dbl(.x, .f, ...)

map_df(.x, .f, ..., .id = NULL)

walk(.x, .f, ...)
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to `.f`.
`.id`	If not `NULL` a variable with this name will be created giving either the name or the index of the data frame.

Details

Note that `map()` understands data frames, including grouped data frames. It can be much faster than `mutate_each()` when your data frame has many columns. However, `map()` will be slower for the more common case of many groups with functions that dplyr knows how to translate to C++.

Value

`map()` always returns a list.

`map_lgl()` returns a logical vector, `map_int()` an integer vector, `map_dbl()`, a double vector, `map_chr()`, a character vector. The output of `.f` will be automatically typed upwards, e.g. logical -> integer -> double -> character.

`walk()` (invisibly) the input `.x`. It's called primarily for its side effects, but this makes it easier to combine in a pipe.

See Also

map2() and pmap() to map over multiple inputs simulatenously

Examples

```
1:10 %>%
  map(rnorm, n = 10) %>%
  map_dbl(mean)

# Or use an anonymous function
1:10 %>%
  map(function(x) rnorm(10, x))

# Or a formula
1:10 %>%
  map(~ rnorm(10, .x))

# A more realistic example: split a data frame into pieces, fit a
# model to each piece, summarise and extract R^2
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map(summary) %>%
  map_dbl("r.squared")

# Use map_lgl(), map_dbl(), etc to reduce to a vector.
# * list
mtcars %>% map(sum)
# * vector
mtcars %>% map_dbl(sum)

# If each element of the output is a data frame, use
# map_df to row-bind them together:
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map_df(~ as.data.frame(t(as.matrix(coef(.)))))
# (if you also want to preserve the variable names see
# the broom package)
```

---

map2 *Map over multiple inputs simultaneously.*

Description

These functions are variants of map() iterate over multiple arguments in parallel. map2 is specialised for the two argument case; pmap allows you to provide any number of arguments in a list.

Usage

```
map2(.x, .y, .f, ...)

map2_lgl(.x, .y, .f, ...)

map2_int(.x, .y, .f, ...)

map2_dbl(.x, .y, .f, ...)

map2_chr(.x, .y, .f, ...)

map2_df(.x, .y, .f, ..., .id = NULL)

pmap(.l, .f, ...)

pmap_lgl(.l, .f, ...)

pmap_int(.l, .f, ...)

pmap_dbl(.l, .f, ...)

pmap_chr(.l, .f, ...)

pmap_df(.l, .f, ..., .id = NULL)

walk2(.x, .y, .f, ...)

pwalk(.l, .f, ...)
```

Arguments

`.x, .y`	Vectors of the same length. A vector of length 1 will be recycled.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to `.f`.
`.id`	If not `NULL` a variable with this name will be created giving either the name or the index of the data frame.
`.l`	A list of lists. The length of `.l` determines the number of arguments that `.f` will be called with. List names will be used if present.

Details

Note that arguments to be vectorised over come before the `.f`, and arguments that are supplied to every call come after `.f`.

`pmap()` and `pwalk()` take a single list `.l` and map over all its elements in parallel.

Value

An atomic vector, list, or data frame, depending on the suffix. Atomic vectors and lists will be named if `.x` or the first element of `.l` is named.

Examples

```
x <- list(1, 10, 100)
y <- list(1, 2, 3)
map2(x, y, ~ .x + .y)
# Or just
map2(x, y, `+`)

# Split into pieces, fit model to each piece, then predict
by_cyl <- mtcars %>% split(.$cyl)
mods <- by_cyl %>% map(~ lm(mpg ~ wt, data = .))
map2(mods, by_cyl, predict)
```

---

negate *Negate a predicate function.*

Description

Negate a predicate function.

Usage

```
negate(.p)
```

Arguments

`.p`	A single predicate function, a formula describing such a predicate function, or a logical vector of the same length as `.x`. Alternatively, if the elements of `.x` are themselves lists of objects, a string indicating the name of a logical element in the inner lists. Only those elements where `.p` evaluates to `TRUE` will be modified.

Value

A new predicate function.

Examples

```
x <- transpose(list(x = 1:10, y = rbernoulli(10)))
x %>% keep("y") %>% length()
x %>% keep(negate("y")) %>% length()
# Same as
x %>% discard("y") %>% length()
```

---

null-default *Default value for* NULL.

Description

This infix function makes it easy to replace NULLs with a default value. It's inspired by the way that Ruby's or operation (||) works.

Usage

```
x %||% y
```

Arguments

x, y	If x is NULL, will return y; otherwise returns x.

Examples

```
1 %||% 2
NULL %||% 2
```

---

partial *Partial apply a function, filling in some arguments.*

Description

Partial function application allows you to modify a function by pre-filling some of the arguments. It is particularly useful in conjunction with functionals and other function operators.

Usage

```
partial(...f, ..., .env = parent.frame(), .lazy = TRUE, .first = TRUE)
```

Arguments

`...f`	a function. For the output source to read well, this should be an be a named function.
`...`	named arguments to `...f` that should be partially applied.
`.env`	the environment of the created function. Defaults to `parent.frame` and you should rarely need to modify this.
`.lazy`	If `TRUE` arguments evaluated lazily, if `FALSE`, evaluated when `partial` is called.
`.first`	If `TRUE`, the partialized arguments are placed to the front of the function signature. If `FALSE`, they are moved to the back. Only useful to control position matching of arguments when the partialized arguments are not named.

Design choices

There are many ways to implement partial function application in R. (see e.g. `dots` in https://github.com/crowding/ptools for another approach.) This implementation is based on creating functions that are as similar as possible to the anonymous functions that you'd create by hand, if you weren't using `partial`.

Examples

```
# Partial is designed to replace the use of anonymous functions for
# filling in function arguments. Instead of:
compact1 <- function(x) discard(x, is.null)

# we can write:
compact2 <- partial(discard, .p = is.null)

# and the generated source code is very similar to what we made by hand
compact1
compact2

# Note that the evaluation occurs "lazily" so that arguments will be
# repeatedly evaluated
f <- partial(runif, n = rpois(1, 5))
f
f()
f()

# You can override this by saying .lazy = FALSE
f <- partial(runif, n = rpois(1, 5), .lazy = FALSE)
f
f()
f()

# This also means that partial works fine with functions that do
# non-standard evaluation
my_long_variable <- 1:10
plot2 <- partial(plot, my_long_variable)
plot2()
plot2(runif(10), type = "l")
```

prepend *Prepend a vector*

Description

This is a companion to append() to help merging two lists or atomic vectors. prepend() is a clearer semantic signal than 'c()' that a vector is to be merged at the beginning of another, especially in a pipe chain.

Usage

```
prepend(x, values, before = 1)
```

Arguments

x	the vector to be modified.
values	to be included in the modified vector.
before	a subscript, before which the values are to be appended.

Value

A merged vector.

Examples

```
x <- as.list(1:3)

x %>% append("a")
x %>% prepend("a")
x %>% prepend(list("a", "b"), before = 3)
```

rbernoulli *Generate random samples from a Bernoulli distribution*

Description

Generate random samples from a Bernoulli distribution

Usage

```
rbernoulli(n, p = 0.5)
```

Arguments

n	Number of samples
p	Probability of getting TRUE

Value

A logical vector

Examples

```
rbernoulli(10)
rbernoulli(100, 0.1)
```

`rdunif` *Generate random samples from a discrete uniform distribution*

Description

Generate random samples from a discrete uniform distribution

Usage

```
rdunif(n, b, a = 1)
```

Arguments

`n`	Number of samples to draw.
`a, b`	Range of the distribution (inclusive).

Examples

```
table(rdunif(1e3, 10))
```

`reduce` *Reduce a list to a single value by iteratively applying a binary function.*

Description

`reduce` combines from the left, `reduce_right` combines from the right.

Usage

```
reduce(.x, .f, ..., .init)

reduce_right(.x, .f, ..., .init)
```

Arguments

.x	A list or atomic vector.
.f	A two-argument function.
...	Additional arguments passed on to .f.
.init	If supplied, will be used as the first value to start the accumulation, rather than using x[[1]]. This is useful if you want to ensure that reduce returns the correct value when .x is is_empty().

Examples

```
1:3 %>% reduce(`+`)
1:10 %>% reduce(`*`)

5 %>%
  replicate(sample(10, 5), simplify = FALSE) %>%
  reduce(intersect)

x <- list(c(0, 1), c(2, 3), c(4, 5))
x %>% reduce(c)
x %>% reduce_right(c)
# Equivalent to:
x %>% rev() %>% reduce(c)

# Use init when you want reduce to return a consistent type when
# given an empty lists
list() %>% reduce(`+`)
list() %>% reduce(`+`, .init = 0)
```

---

rerun *Re-run expressions multiple times.*

Description

This is a convenient way of generating sample data. It works similarly to replicate(..., simplify = FALSE).

Usage

```
rerun(.n, ...)
```

Arguments

.n	Number of times to run expressions
...	Expressions to re-run.

Value

A list of length `.n`. Each element of `...` will be re-run once for each `.n`. It

There is one special case: if there's a single unnamed input, the second level list will be dropped. In this case, `rerun(n, x)` behaves like `replicate(n, x, simplify = FALSE)`.

Examples

```
10 %>% rerun(rnorm(5))
10 %>%
  rerun(x = rnorm(5), y = rnorm(5)) %>%
  map_dbl(~ cor(.x$x, .x$y))
```

`safely` *Capture side effects.*

Description

These functions wrap functions so instead generating side effects through output, messages, warnings, and errors, they instead return enchanced output. They are all adverbs because they modify the action of a verb (a function).

Usage

```
safely(.f, otherwise = NULL, quiet = TRUE)

quietly(.f)

possibly(.f, otherwise, quiet = TRUE)
```

Arguments

`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`otherwise`	Default value to use when an error occurs.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Value

`safe`: a list with components `result` and `error`. One value is always `NULL`

`outputs`: a list with components `result`, `output`, `messages` and `warnings`.

Examples

```
safe_log <- safely(log)
safe_log(10)
safe_log("a")

list("a", 10, 100) %>%
  map(safe_log) %>%
  transpose()

# This is a bit easier to work with if you supply a default value
# of the same type and use the simplify argument to transpose():
safe_log <- safely(log, otherwise = NA_real_)
list("a", 10, 100) %>%
  map(safe_log) %>%
  transpose() %>%
  simplify_all()

# To replace errors with a default value, use possibly().
list("a", 10, 100) %>%
  map_dbl(possibly(log, NA_real_))
```

---

scalar-type-predicates

Scalar type predicates

Description

These predicates check for a given type and whether the vector is "scalar", that is, of length 1.

Usage

```
is_scalar_list(x)

is_scalar_atomic(x)

is_scalar_vector(x)

is_scalar_numeric(x)

is_scalar_integer(x)

is_scalar_double(x)

is_scalar_character(x)

is_scalar_logical(x)
```

Arguments

x	object to be tested.

See Also

type-predicates bare-type-predicates

---

set_names *Set names in a vector*

Description

This is a snake case wrapper for setNames, with tweaked defaults, and stricter argument checking.

Usage

```
set_names(x, nm = x)
```

Arguments

x	Vector to name
nm	Vector of names, the same length as x

Examples

```
set_names(1:4, c("a", "b", "c", "d"))

# If the second argument is ommitted a vector is named with itself
set_names(letters[1:5])
```

---

splice *Splice objects and lists of objects into a list*

Description

This splices all arguments into a list. Non-list objects and lists with a S3 class are encapsulated in a list before concatenation.

Usage

```
splice(...)
```

Arguments

...	Objects to concatenate.

Examples

```
inputs <- list(arg1 = "a", arg2 = "b")

# splice() concatenates the elements of inputs with arg3
splice(inputs, arg3 = c("c1", "c2")) %>% str()
list(inputs, arg3 = c("c1", "c2")) %>% str()
c(inputs, arg3 = c("c1", "c2")) %>% str()
```

`split_by` *Split, order and sort lists by their components.*

Description

Split, order and sort lists by their components.

Usage

```
split_by(.x, .f, ...)

order_by(.x, .f, ...)

sort_by(.x, .f, ...)
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or atomic vector. If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function with two arguments, `.x` or `.` and `.y`. This allows you to create very compact anonymous functions with up to two inputs. If **character** or **integer vector**, e.g. `"y"`, it is converted to an extractor function, `function(x) x[["y"]]`. To index deeply into a nested list, use multiple values; `c("x", "y")` is equivalent to `z[["x"]][["y"]]`. You can also set `.null` to set a default to use instead of `NULL` for absent components.
`...`	Additional arguments passed on to `.f`.

Examples

```
l1 <- transpose(list(x = sample(10), y = 1:10))
l1
l1 %>% order_by("x")
l1 %>% sort_by("x")

l2 <- rerun(5, g = sample(2, 1), y = rdunif(5, 10))
l2 %>% split_by("g") %>% str()
l2 %>% split_by("g") %>% map(. %>% map("y"))
```

`transpose` *Transpose a list.*

Description

Tranpose turns a list-of-lists "inside-out"; it turns a pair of lists into a list of pairs, or a list of pairs into pair of lists. For example, If you had a list of length n where each component had values a and b, `transpose()` would make a list with elements a and b that contained lists of length n. It's called transpose because `x[[1]][[2]]` is equivalent to `transpose(x)[[2]][[1]]`.

Usage

```
transpose(.l)
```

Arguments

`.l`	A list of vectors to zip. The first element is used as the template; you'll get a warning if a sub-list is not the same length as the first element. For efficiency, elements are matched by position, not by name.

Details

Note that `transpose()` is its own inverse, much like the transpose operation on a matrix. You can get back the original input by transposing it twice.

Examples

```
x <- rerun(5, x = runif(1), y = runif(5))
x %>% str()
x %>% transpose() %>% str()
# Back to where we started
x %>% transpose() %>% transpose() %>% str()

# transpose() is useful in conjunction with safely() & quietly()
x <- list("a", 1, 2)
y <- x %>% map(safely(log))
y %>% str()
y %>% transpose() %>% str()

# Use simplify_all() to reduce to atomic vectors where possible
x <- list(list(a = 1, b = 2), list(a = 3, b = 4), list(a = 5, b = 6))
x %>% transpose()
x %>% transpose() %>% simplify_all()
```

type-predicates *Type predicates*

Description

These type predicates aim to make type testing in R more consistent. They are wrappers around `typeof`, so operate at a level beneath S3/S4 etc.

Usage

```
is_list(x)

is_atomic(x)

is_vector(x)

is_numeric(x)

is_integer(x)

is_double(x)

is_character(x)

is_logical(x)

is_null(x)

is_function(x)
```

Arguments

x	object to be tested.

Details

Compare to base R functions:

- Unlike `is.atomic()`, `is_atomic()` does not return `TRUE` for `NULL`.
- Unlike `is.vector()`, `is_vector()` test if an object is an atomic vector or a list. `is.vector` checks for the presence of attributes (other than name).
- `is_numeric()` is not generic so, (e.g.) dates and date times are `TRUE`, not `FALSE`.
- `is_function()` returns `TRUE` only for regular functions, not special or primitive functions.

See Also

bare-type-predicates scalar-type-predicates

update_list *Modify a list*

Description

Modify a list

Usage

```
update_list(`_data`, ...)
```

Arguments

_data	A list.
...	New values of a list. Use NULL to remove values. Use a formula to evaluate in the context of the list values.

Examples

```
x <- list(x = 1:10, y = 4)
update_list(x, z = 10)
update_list(x, z = ~ x + y)
```

when *Match/validate a set of conditions for an object and continue with the action associated with the first valid match.*

Description

when is a flavour of pattern matching (or an if-else abstraction) in which a value is matched against a sequence of condition-action sets. When a valid match/condition is found the action is executed and the result of the action is returned.

Usage

```
when(., ...)
```

Arguments

.	the value to match agaist
...	formulas; each containing a condition as LHS and an action as RHS. named arguments will define additional values.

Value

The value resulting from the action of the first valid match/condition is returned. If no matches are found, and no default is given, NULL will be returned.

Validity of the conditions are tested with `isTRUE`, or equivalently with `identical(condition, TRUE)`. In other words conditions resulting in more than one logical will never be valid. Note that the input value is always treated as a single object, as opposed to the `ifelse` function.

Examples

```
1:10 %>%
  when(
    sum(.) <=  50 ~ sum(.),
    sum(.) <= 100 ~ sum(.)/2,
    ~ 0
  )

1:10 %>%
  when(
    sum(.) <=   x ~ sum(.),
    sum(.) <= 2*x ~ sum(.)/2,
    ~ 0,
    x = 60
  )

iris %>%
  subset(Sepal.Length > 10) %>%
  when(
    nrow(.) > 0 ~ .,
    ~ iris %>% head(10)
  )

iris %>%
  head %>%
  when(nrow(.) < 10 ~ .,
       ~ stop("Expected fewer than 10 rows."))
```

Index