Package 'purrr'

April 16, 2020

Title Functional Programming Tools

Version 0.3.4

Description A complete and consistent functional programming toolkit for R.

License GPL-3 | file LICENSE

URL http://purrr.tidyverse.org, https://github.com/tidyverse/purrr

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

Depends R (>= 3.2)

Imports magrittr (>= 1.5), rlang (>= 0.3.1)

Suggests covr, crayon, dplyr (>= 0.7.8), knitr, rmarkdown, testthat, tibble, tidyselect

VignetteBuilder knitr

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.1.0

R topics documented:

`accumulate` *Accumulate intermediate results of a vector reduction*

Description

`accumulate()` sequentially applies a 2-argument function to elements of a vector. Each application of the function uses the initial value or result of the previous application as the first argument. The second argument is the next value of the vector. The results of each application are returned in a list. The accumulation can optionally terminate before processing the whole vector in response to a `done()` signal returned by the accumulation function.

By contrast to `accumulate()`, `reduce()` applies a 2-argument function in the same way, but discards all results except that of the final function application.

`accumulate2()` sequentially applies a function to elements of two lists, `.x` and `.y`.

Usage

```
accumulate(.x, .f, ..., .init, .dir = c("forward", "backward"))

accumulate2(.x, .y, .f, ..., .init)
```

Arguments

`.x`	A list or atomic vector.
`.f`	For `accumulate()` `.f` is 2-argument function. The function will be passed the accumulated result or initial value as the first argument. The next value in sequence is passed as the second argument. For `accumulate2()`, a 3-argument function. The function will be passed the accumulated result as the first argument. The next value in sequence from `.x` is passed as the second argument. The next value in sequence from `.y` is passed as the third argument. The accumulation terminates early if `.f` returns a value wrapped in a `done()`.
`...`	Additional arguments passed on to the mapped function.
`.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 a correct value when `.x` is empty. If missing, and `.x` is empty, will throw an error.
`.dir`	The direction of accumulation as a string, one of `"forward"` (the default) or `"backward"`. See the section about direction below.
`.y`	For `accumulate2()` `.y` is the second argument of the pair. It needs to be 1 element shorter than the vector to be accumulated (`.x`). If `.init` is set, `.y` needs to be one element shorted than the concatenation of the initial value and `.x`.

Value

A vector the same length of `.x` with the same names as `.x`.

If `.init` is supplied, the length is extended by 1. If `.x` has names, the initial value is given the name `".init"`, otherwise the returned vector is kept unnamed.

If `.dir` is `"forward"` (the default), the first element is the initial value (`.init` if supplied, or the first element of `.x`) and the last element is the final reduced value. In case of a right accumulation, this order is reversed.

The accumulation terminates early if `.f` returns a value wrapped in a `done()`. If the done box is empty, the last value is used instead and the result is one element shorter (but always includes the initial value, even when terminating at the first iteration).

Life cycle

`accumulate_right()` is soft-deprecated in favour of the `.dir` argument as of rlang 0.3.0. Note that the algorithm has slightly changed: the accumulated value is passed to the right rather than the left, which is consistent with a right reduction.

Direction

When `.f` is an associative operation like `+` or `c()`, the direction of reduction does not matter. For instance, reducing the vector `1:3` with the binary function `+` computes the sum `((1 + 2) + 3)` from the left, and the same sum `(1 + (2 + 3))` from the right.

In other cases, the direction has important consequences on the reduced value. For instance, reducing a vector with `list()` from the left produces a left-leaning nested list (or tree), while reducing `list()` from the right produces a right-leaning list.

See Also

`reduce()` when you only need the final reduced value.

Examples

```
# With an associative operation, the final value is always the
# same, no matter the direction. You'll find it in the last element for a
# backward (left) accumulation, and in the first element for forward
# (right) one:
1:5 %>% accumulate(`+`)
1:5 %>% accumulate(`+`, .dir = "backward")

# The final value is always equal to the equivalent reduction:
1:5 %>% reduce(`+`)

# It is easier to understand the details of the reduction with
# `paste()`.
accumulate(letters[1:5], paste, sep = ".")

# Note how the intermediary reduced values are passed to the left
# with a left reduction, and to the right otherwise:
accumulate(letters[1:5], paste, sep = ".", .dir = "backward")

# `accumulate2()` is a version of `accumulate()` that works with
# 3-argument functions and one additional vector:
paste2 <- function(x, y, sep = ".") paste(x, y, sep = sep)
letters[1:4] %>% accumulate(paste2)
letters[1:4] %>% accumulate2(c("-", ".", "-"), paste2)


# You can shortcircuit an accumulation and terminate it early by
# returning a value wrapped in a done(). In the following example
# we return early if the result-so-far, which is passed on the LHS,
# meets a condition:
paste3 <- function(out, input, sep = ".") {
  if (nchar(out) > 4) {
    return(done(out))
  }
  paste(out, input, sep = sep)
}
letters %>% accumulate(paste3)

# Note how we get twice the same value in the accumulation. That's
# because we have returned it twice. To prevent this, return an empty
# done box to signal to accumulate() that it should terminate with the
# value of the last iteration:
paste3 <- function(out, input, sep = ".") {
```

```
  if (nchar(out) > 4) {
    return(done())
  }
  paste(out, input, sep = sep)
}
letters %>% accumulate(paste3)

# Here the early return branch checks the incoming inputs passed on
# the RHS:
paste4 <- function(out, input, sep = ".") {
  if (input == "f") {
    return(done())
  }
  paste(out, input, sep = sep)
}
letters %>% accumulate(paste4)


# 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_dfr(~ tibble(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)
```

`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_mapper *Convert an object into a mapper function*

Description

`as_mapper` is the powerhouse behind the varied function specifications that most purrr functions allow. It is an S3 generic. The default method forwards its arguments to `rlang::as_function()`.

Usage

```
as_mapper(.f, ...)

## S3 method for class 'character'
as_mapper(.f, ..., .null, .default = NULL)

## S3 method for class 'numeric'
as_mapper(.f, ..., .null, .default = NULL)

## S3 method for class 'list'
as_mapper(.f, ..., .null, .default = NULL)
```

Arguments

`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to methods.
`.default, .null`	Optional additional argument for extractor functions (i.e. when `.f` is character, integer, or list). Returned when value is absent (does not exist) or empty (has length 0). `.null` is deprecated; please use `.default` instead.

Examples

```
as_mapper(~ . + 1)
as_mapper(1)

as_mapper(c("a", "b", "c"))
# Equivalent to function(x) x[["a"]][["b"]][["c"]]

as_mapper(list(1, "a", 2))
# Equivalent to function(x) x[[1]][["a"]][[2]]

as_mapper(list(1, attr_getter("a")))
# Equivalent to function(x) attr(x[[1]], "a")

as_mapper(c("a", "b", "c"), .default = 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
```

`attr_getter` *Create an attribute getter function*

Description

`attr_getter()` generates an attribute accessor function; i.e., it generates a function for extracting an attribute with a given name. Unlike the base R `attr()` function with default options, it doesn't use partial matching.

Usage

```
attr_getter(attr)
```

Arguments

`attr`	An attribute name as string.

See Also

`pluck()`

Examples

```
# attr_getter() takes an attribute name and returns a function to
# access the attribute:
get_rownames <- attr_getter("row.names")
get_rownames(mtcars)

# These getter functions are handy in conjunction with pluck() for
# extracting deeply into a data structure. Here we'll first
# extract by position, then by attribute:
obj1 <- structure("obj", obj_attr = "foo")
obj2 <- structure("obj", obj_attr = "bar")
x <- list(obj1, obj2)

pluck(x, 1, attr_getter("obj_attr"))  # From first object
pluck(x, 2, attr_getter("obj_attr"))  # From second object
```

`compose` *Compose multiple functions*

Description

Compose multiple functions

Usage

```
compose(..., .dir = c("backward", "forward"))
```

Arguments

`...`	Functions to apply in order (from right to left by default). Formulas are converted to functions in the usual way. These dots support tidy dots features. In particular, if your functions are stored in a list, you can splice that in with !!!.
`.dir`	If `"backward"` (the default), the functions are called in the reverse order, from right to left, as is conventional in mathematics. If `"forward"`, they are called from left to right.

Value

A function

Examples

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

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

# You can use the formula shortcut for functions:
fn <- compose(~ paste(.x, "foo"), ~ paste(.x, "bar"))
fn("input")
```

```
# Lists of functions can be spliced with !!!
fns <- list(
  function(x) paste(x, "foo"),
  ~ paste(.x, "bar")
)
fn <- compose(!!!fns)
fn("input")
```

`cross` *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()` takes a list `.l` and returns the cartesian product of all its elements in a list, with one combination by element. `cross_df()` is like `cross()` but returns a data frame, with one combination by row.

Usage

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

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

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

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

Arguments

`.l`	A list of lists or atomic vectors. Alternatively, a data frame. `cross_df()` 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()`, `cross2()` and `cross3()` return the cartesian product is returned in wide format. This makes it more amenable to mapping operations. `cross_df()` 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()`) and returns `TRUE` or `FALSE`. The combinations where the predicate function returns `TRUE` will be removed from the result.

Value

`cross2()`, `cross3()` and `cross()` always return a list. `cross_df()` always returns a data frame. `cross()` returns a list where each element is one combination so that the list can be directly mapped over. `cross_df()` 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() %>%
  map(lift(paste))

# cross() returns the combinations in long format: many elements,
# each representing one combination. With cross_df() 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_df()

# In case you need a list in long format (and not a data frame)
# just run as.list() after cross_df()
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(.filter = `==`)
```

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

Description

Find the value or position of the first match

Usage

```
detect(
  .x,
  .f,
  ...,
  .dir = c("forward", "backward"),
  .right = NULL,
  .default = NULL
)

detect_index(.x, .f, ..., .dir = c("forward", "backward"), .right = NULL)
```

Arguments

.x	A list or atomic vector.
.f	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
...	Additional arguments passed on to the mapped function.
.dir	If `"forward"`, the default, starts at the beginning of the vector and move towards the end; if `"backward"`, starts at the end of the vector and moves towards the beginning.
.right	Soft-deprecated. Please use `.dir` instead.
.default	The value returned when nothing is detected.

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.

See Also

keep() for keeping all matching values.

Examples

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

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

3:10 %>% detect(is_even, .dir = "backward")
3:10 %>% detect_index(is_even, .dir = "backward")


# Since `.f` is passed to as_mapper(), you can supply a
# lambda-formula or a pluck object:
x <- list(
  list(1, foo = FALSE),
  list(2, foo = TRUE),
  list(3, foo = TRUE)
)

detect(x, "foo")
detect_index(x, "foo")


# If you need to find all values, use keep():
keep(x, "foo")

# If you need to find all positions, use map_lgl():
which(map_lgl(x, "foo"))
```

---

done *Done box*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

rlang done

---

every *Do every, some, or none of the elements of a list satisfy a predicate?*

Description

Do every, some, or none of the elements of a list satisfy a predicate?

Usage

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

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

none(.x, .p, ...)
```

Arguments

.x	A list or atomic vector.
.p	A predicate function to apply on each element of .x. some() returns TRUE when .p is TRUE for at least one element. every() returns TRUE when .p is TRUE for all elements. none() returns TRUE when .p is FALSE for all elements.‘
...	Additional arguments passed on to .p.

Value

A logical vector of length 1.

Examples

```
y <- list(0:10, 5.5)
y %>% every(is.numeric)
y %>% every(is.integer)
y %>% some(is.integer)
y %>% none(is.character)
```

---

exec *Execute a function*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

rlang exec

---

faq-adverbs-export *Best practices for exporting adverb-wrapped functions*

Description

Functions like insistently(), safely(), slowly(), and quietly() help resolve challenging issues in programming. For example, safely() modifies a function to return both an error and a result. These functions work by returning an enhanced version of the original function. They are often called **adverb** functions and are typically named with an informative prefix such as safe_ or insist_. For instance, an insistent variant of scrape_data() created with insistently() might be called insist_scrape_data().

Exporting functions created with purrr adverbs in your package requires some precautions. Because the functions created by adverbs contain internal purrr code, creating them once and for all when

the package is built might cause problems when purrr is updated. Instead, the function must be created by the purrr adverb each time the package is loaded in memory via the `.onLoad()` hook of the package. This prevents the generated functions to contain outdated internal purrr code (which could even refer to functions that no longer exist in the purrr namespace).

An example if provided below for `insist`, but it would be very similar for functions generated by other adverbs.

```
#' My function
insist_my_function <- function(...) "dummy"

my_function <- function(...) {
  # Implementation
}

.onLoad <- function(lib, pkg) {
  insist_my_function <<- purrr::insistently(my_function)
}
```

`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()`, but they only ever remove a single layer of hierarchy and they 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_raw(.x)

flatten_dfr(.x, .id = NULL)

flatten_dfc(.x)
```

Arguments

`.x`	A list to 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	Either a string or NULL. If a string, the output will contain a variable with that name, storing either the name (if .x is named) or the index (if .x is unnamed) of the input. If NULL, the default, no variable will be created. Only applies to _dfr variant.

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.

flatten_dfr() and flatten_dfc() return data frames created by row-binding and column-binding respectively. They require dplyr to be installed.

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)
```

---

has_element *Does a list contain an object?*

Description

Does a list contain an object?

Usage

```
has_element(.x, .y)
```

Arguments

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

Examples

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

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 the mapped function.

Value

A vector the same type as .x.

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)
```

imap *Apply a function to each element of a vector, and its index*

Description

imap_xxx(x,...), an indexed map, is short hand for map2(x,names(x),...) if x has names, or map2(x,seq_along(x),...) if it does not. This is useful if you need to compute on both the value and the position of an element.

Usage

```
imap(.x, .f, ...)

imap_lgl(.x, .f, ...)

imap_chr(.x, .f, ...)

imap_int(.x, .f, ...)

imap_dbl(.x, .f, ...)

imap_raw(.x, .f, ...)

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

imap_dfc(.x, .f, ...)

iwalk(.x, .f, ...)
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.id`	Either a string or `NULL`. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If `NULL`, the default, no variable will be created. Only applies to _dfr variant.

Value

A vector the same length as `.x`.

See Also

Other map variants: `invoke()`, `lmap()`, `map2()`, `map_if()`, `map()`, `modify()`

Examples

```
# Note that when using the formula shortcut, the first argument
# is the value, and the second is the position
imap_chr(sample(10), ~ paste0(.y, ": ", .x))
iwalk(mtcars, ~ cat(.y, ": ", median(.x), "\n", sep = ""))
```

---

insistently *Transform a function to make it run insistently or slowly*

Description

- `insistently()` takes a function and modifies it to retry a given amount of time on error.
- `slowly()` takes a function and modifies it to wait a given amount of time between each call.

The number and rate of attempts is determined by a rate object (by default a jittered exponential backoff rate for `insistently()`, and a constant rate for `slowly()`).

If you would like to include a function created with `safely`, `slowly`, or `insistently` in a package, see faq-adverbs-export.

Usage

```
insistently(f, rate = rate_backoff(), quiet = TRUE)

slowly(f, rate = rate_delay(), quiet = TRUE)
```

Arguments

`f`	A function to modify.
`rate`	A rate object determining the waiting time.
`quiet`	If `FALSE`, prints a message displaying how long until the next request.

See Also

`httr::RETRY()` is a special case of `insistently()` for HTTP verbs. `rate_backoff()` and `rate_delay()` for creating custom backoff rates. `rate_sleep()` for the function powering `insistently()` and `slowly()`. `safely()` for another useful function operator.

Examples

```
# For the purpose of this example, we first create a custom rate
# object with a low waiting time between attempts:
rate <- rate_delay(0.1)

# slowly() causes a function to sleep for a given time between calls:
slow_runif <- slowly(~ runif(1), rate = rate, quiet = FALSE)
map(1:5, slow_runif)


# insistently() makes a function repeatedly try to work
risky_runif <- function(lo = 0, hi = 1) {
  y <- runif(1, lo, hi)
  if(y < 0.9) {
```

```
    stop(y, " is too small")
  }
  y
}

# Let's now create an exponential backoff rate with a low waiting
# time between attempts:
rate <- rate_backoff(pause_base = 0.1, pause_min = 0.005, max_times = 4)

# Modify your function to run insistently.
insistent_risky_runif <- insistently(risky_runif, rate, quiet = FALSE)

set.seed(6) # Succeeding seed
insistent_risky_runif()

set.seed(3) # Failing seed
try(insistent_risky_runif())


# You can also use other types of rate settings, like a delay rate
# that waits for a fixed amount of time. Be aware that a delay rate
# has an infinite amount of attempts by default:
rate <- rate_delay(0.2, max_times = 3)
insistent_risky_runif <- insistently(risky_runif, rate = rate, quiet = FALSE)
try(insistent_risky_runif())


# insistently() and possibly() are a useful combination
rate <- rate_backoff(pause_base = 0.1, pause_min = 0.005)
possibly_insistent_risky_runif <- possibly(insistent_risky_runif, otherwise = -99)

set.seed(6)
possibly_insistent_risky_runif()

set.seed(3)
possibly_insistent_risky_runif()
```

---

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

Description

keep() and discard() are opposites. compact() is a handy wrapper that removes all empty elements.

Usage

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

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

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

Arguments

`.x`	A list or vector.
`.p`	For `keep()` and `discard()`, a predicate function. Only those elements where `.p` evaluates to `TRUE` will be kept or discarded. For `compact()`, a function that is applied to each element of `.x`. Only those elements where `.p` evaluates to an empty vector will be discarded.
`...`	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 the predicate function `.p` 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")

# compact() discards elements that are NULL or that have length zero
list(a = "a", b = NULL, c = integer(0), d = NA, e = list()) %>%
  compact()
```

`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.

Value

A function.

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() 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_dots <- lift_ld(cross)
out1 <- cross(list(a = 1:2, b = c("a", "b", "c")))
out2 <- cross_dots(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, .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
```

`list_modify` *Modify a list*

Description

`list_modify()` and `list_merge()` recursively combine two lists, matching elements either by name or position. If a sub-element is present in both lists `list_modify()` takes the value from `y`, and `list_merge()` concatenates the values together.

`update_list()` handles formulas and quosures that can refer to values existing within the input list. Note that this function might be deprecated in the future in favour of a `dplyr::mutate()` method for lists.

Usage

```
list_modify(.x, ...)

list_merge(.x, ...)
```

Arguments

`.x`	List to modify.
`...`	New values of a list. Use `zap()` to remove values. These values should be either all named or all unnamed. When inputs are all named, they are matched to `.x` by name. When they are all unnamed, they are matched positionally. These dots support tidy dots features. In particular, if your functions are stored in a list, you can splice that in with !!!.

Examples

```
x <- list(x = 1:10, y = 4, z = list(a = 1, b = 2))
str(x)

# Update values
str(list_modify(x, a = 1))
# Replace values
str(list_modify(x, z = 5))
str(list_modify(x, z = list(a = 1:5)))

# Remove values
str(list_modify(x, z = zap()))

# Combine values
str(list_merge(x, x = 11, z = list(a = 2:5, c = 3)))


# All these functions support tidy dots features. Use !!! to splice
# a list of arguments:
l <- list(new = 1, y = zap(), z = 5)
str(list_modify(x, !!!l))
```

`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 elements `list-elements`).

Usage

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

lmap_if(.x, .p, .f, ..., .else = NULL)

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 the mapped function.
.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.
.else	A function applied to elements of .x for which .p returns FALSE.
.at	A character vector of names, positive numeric vector of positions to include, or a negative numeric vector of positions to exlude. Only those elements corresponding to .at will be modified. If the tidyselect package is installed, you can use vars() and the tidyselect helpers to select elements.

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

Other map variants: imap(), invoke(), map2(), map_if(), map(), modify()

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)
  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 list or atomic vector*

Description

The map functions transform their input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input.

- `map()` always returns a list. See the `modify()` family for versions that return an object of the same type as the input.
- `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()` return an atomic vector of the indicated type (or die trying).
- `map_dfr()` and `map_dfc()` return a data frame created by row-binding and column-binding respectively. They require dplyr to be installed.
- The returned values of `.f` must be of length one for each element of `.x`. If `.f` uses an extractor function shortcut, `.default` can be specified to handle values that are absent or empty. See `as_mapper()` for more on `.default`.
- `walk()` calls `.f` for its side-effect and returns the input `.x`.

Usage

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

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

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

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

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

map_raw(.x, .f, ...)

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

map_dfc(.x, .f, ...)

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

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.id`	Either a string or `NULL`. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If `NULL`, the default, no variable will be created. Only applies to _dfr variant.

Value

- `map()` Returns a list the same length as `.x`.
- `map_lgl()` returns a logical vector, `map_int()` an integer vector, `map_dbl()` a double vector, and `map_chr()` a character vector.
- `map_df()`, `map_dfc()`, `map_dfr()` all return a data frame.
- If `.x` has `names()`, the return value preserves those names.
- The output of `.f` will be automatically typed upwards, e.g. logical -> integer -> double -> character.
- `walk()` returns the input `.x` (invisibly). This makes it easy to use in pipe.

See Also

`map_if()` for applying a function to only those elements of `.x` that meet a specified condition.

Other map variants: `imap()`, `invoke()`, `lmap()`, `map2()`, `map_if()`, `modify()`

Examples

```
# Compute normal distributions from an atomic vector
1:10 %>%
  map(rnorm, n = 10)

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

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

# Simplify output to a vector instead of a list by computing the mean of the distributions
1:10 %>%
  map(rnorm, n = 10) %>%  # output a list
  map_dbl(mean)           # output an atomic vector

# Using set_names() with character vectors is handy to keep track
# of the original inputs:
set_names(c("foo", "bar")) %>% map_chr(paste0, ":suffix")

# Working with lists
favorite_desserts <- list(Sophia = "banana bread", Eliott = "pancakes", Karina = "chocolate cake")
favorite_desserts %>% map_chr(~ paste(.x, "rocks!"))

# Extract by name or position
# .default specifies value for elements that are missing or NULL
l1 <- list(list(a = 1L), list(a = NULL, b = 2L), list(b = 3L))
l1 %>% map("a", .default = "???")
l1 %>% map_int("b", .default = NA)
l1 %>% map_int(2, .default = NA)

# Supply multiple values to index deeply into a list
l2 <- list(
  list(num = 1:3,     letters[1:3]),
  list(num = 101:103, letters[4:6]),
  list()
)
l2 %>% map(c(2, 2))

# Use a list to build an extractor that mixes numeric indices and names,
# and .default to provide a default value if the element does not exist
l2 %>% map(list("num", 3))
l2 %>% map_int(list("num", 3), .default = NA)

# Working with data frames
# Use map_lgl(), map_dbl(), etc to return a vector instead of a list:
mtcars %>% map_dbl(sum)

# 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")

# If each element of the output is a data frame, use
# map_dfr to row-bind them together:
mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map_dfr(~ 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()` that iterate over multiple arguments simultaneously. They are parallel in the sense that each input is processed in parallel with the others, not in the sense of multicore computing. They share the same notion of "parallel" as `base::pmax()` and `base::pmin()`. `map2()` and `walk2()` are specialised for the two argument case; `pmap()` and `pwalk()` allow you to provide any number of arguments in a list. Note that a data frame is a very important special case, in which case `pmap()` and `pwalk()` apply the function `.f` to each row. `map_dfr()`, `pmap_dfr()` and `map2_dfc()`, `pmap_dfc()` return data frames created by row-binding and column-binding respectively. They require dplyr to be installed.

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_raw(.x, .y, .f, ...)

map2_dfr(.x, .y, .f, ..., .id = NULL)

map2_dfc(.x, .y, .f, ...)

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

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

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

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

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

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

pmap_raw(.l, .f, ...)

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

pmap_dfc(.l, .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 vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.id`	Either a string or `NULL`. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If `NULL`, the default, no variable will be created. Only applies to _dfr variant.
`.l`	A list of vectors, such as a data frame. 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 `.f`, and arguments that are supplied to every call come after `.f`.

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.

If all input is length 0, the output will be length 0. If any input is length 1, it will be recycled to the length of the longest.

See Also

Other map variants: imap(), invoke(), lmap(), map_if(), map(), modify()

Examples

```
x <- list(1, 1, 1)
y <- list(10, 20, 30)
z <- list(100, 200, 300)

map2(x, y, ~ .x + .y)
# Or just
map2(x, y, `+`)

pmap(list(x, y, z), sum)

# Matching arguments by position
pmap(list(x, y, z), function(first, second, third) (first + third) * second)

# Matching arguments by name
l <- list(a = x, b = y, c = z)
pmap(l, function(c, b, a) (a + c) * b)

# 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)

# Vectorizing a function over multiple arguments
df <- data.frame(
  x = c("apple", "banana", "cherry"),
  pattern = c("p", "n", "h"),
  replacement = c("P", "N", "H"),
  stringsAsFactors = FALSE
  )
pmap(df, gsub)
pmap_chr(df, gsub)

# Use `...` to absorb unused components of input list .l
df <- data.frame(
  x = 1:3,
  y = 10:12,
  z = letters[1:3]
)
plus <- function(x, y) x + y
## Not run:
# this won't work
pmap(df, plus)

## End(Not run)
# but this will
plus2 <- function(x, y, ...) x + y
pmap_dbl(df, plus2)

# The "p" for "parallel" in pmap() is the same as in base::pmin()
# and base::pmax()
df <- data.frame(
```

```
  x = c(1, 2, 5),
  y = c(5, 4, 8)
)
# all produce the same result
pmin(df$x, df$y)
map2_dbl(df$x, df$y, min)
pmap_dbl(df, min)

# If you want to bind the results of your function rowwise, use:
# map2_dfr() or pmap_dfr()
ex_fun <- function(arg1, arg2){
col <- arg1 + arg2
x <- as.data.frame(col)
}
arg1 <- 1:4
arg2 <- 10:13
map2_dfr(arg1, arg2, ex_fun)
# If instead you want to bind by columns, use map2_dfc() or pmap_dfc()
map2_dfc(arg1, arg2, ex_fun)
```

`map_if` *Apply a function to each element of a vector conditionally*

Description

The functions `map_if()` and `map_at()` take `.x` as input, apply the function `.f` to some of the elements of `.x`, and return a list of the same length as the input.

- `map_if()` takes a predicate function `.p` as input to determine which elements of `.x` are transformed with `.f`.
- `map_at()` takes a vector of names or positions `.at` to specify which elements of `.x` are transformed with `.f`.
- `map_depth()` allows to apply `.f` to a specific depth level of a nested vector.

Usage

```
map_if(.x, .p, .f, ..., .else = NULL)

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

map_depth(.x, .depth, .f, ..., .ragged = 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.

`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.else`	A function applied to elements of `.x` for which `.p` returns `FALSE`.
`.at`	A character vector of names, positive numeric vector of positions to include, or a negative numeric vector of positions to exlude. Only those elements corresponding to `.at` will be modified. If the `tidyselect` package is installed, you can use `vars()` and the `tidyselect` helpers to select elements.
`.depth`	Level of `.x` to map on. Use a negative value to count up from the lowest level of the list. • `map_depth(x,0,fun)` is equivalent to `fun(x)`. • `map_depth(x,1,fun)` is equivalent to `x <-map(x,fun)` • `map_depth(x,2,fun)` is equivalent to `x <-map(x,~ map(.,fun))`
`.ragged`	If `TRUE`, will apply to leaves, even if they're not at depth `.depth`. If `FALSE`, will throw an error if there are no elements at depth `.depth`.

See Also

Other map variants: `imap()`, `invoke()`, `lmap()`, `map2()`, `map()`, `modify()`

Examples

```
# Use a predicate function to decide whether to map a function:
map_if(iris, is.factor, as.character)

# Specify an alternative with the `.else` argument:
map_if(iris, is.factor, as.character, .else = as.integer)

# Use numeric vector of positions select elements to change:
iris %>% map_at(c(4, 5), is.numeric)

# Use vector of names to specify which elements to change:
iris %>% map_at("Species", toupper)

# Use `map_depth()` to recursively traverse nested vectors and map
# a function at a certain depth:
x <- list(a = list(foo = 1:2, bar = 3:4), b = list(baz = 5:6))
str(x)
map_depth(x, 2, paste, collapse = "/")

# Equivalent to:
map(x, map, paste, collapse = "/")
```

`modify` *Modify elements selectively*

Description

Unlike `map()` and its variants which always return a fixed object type (list for `map()`, integer vector for `map_int()`, etc), the `modify()` family always returns the same type as the input object.

- `modify()` is a shortcut for x[[i]] <- f(x[[i]]); return(x).
- `modify_if()` only modifies the elements of `x` that satisfy a predicate and leaves the others unchanged. `modify_at()` only modifies elements given by names or positions.
- `modify2()` modifies the elements of `.x` but also passes the elements of `.y` to `.f`, just like `map2()`. `imodify()` passes the names or the indices to `.f` like `imap()` does.
- `modify_depth()` only modifies elements at a given level of a nested data structure.
- `modify_in()` modifies a single element in a `pluck()` location.

Usage

```
modify(.x, .f, ...)

## Default S3 method:
modify(.x, .f, ...)

modify_if(.x, .p, .f, ..., .else = NULL)

## Default S3 method:
modify_if(.x, .p, .f, ..., .else = NULL)

modify_at(.x, .at, .f, ...)

## Default S3 method:
modify_at(.x, .at, .f, ...)

modify2(.x, .y, .f, ...)

imodify(.x, .f, ...)

modify_depth(.x, .depth, .f, ..., .ragged = .depth < 0)

## Default S3 method:
modify_depth(.x, .depth, .f, ..., .ragged = .depth < 0)
```

Arguments

`.x`	A list or atomic vector.
`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions.

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.

`...` Additional arguments passed on to the mapped function.

`.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.

`.else` A function applied to elements of `.x` for which `.p` returns `FALSE`.

`.at` A character vector of names, positive numeric vector of positions to include, or a negative numeric vector of positions to exlude. Only those elements corresponding to `.at` will be modified. If the `tidyselect` package is installed, you can use `vars()` and the `tidyselect` helpers to select elements.

`.y` Vectors of the same length. A vector of length 1 will be recycled.

`.depth` Level of `.x` to map on. Use a negative value to count up from the lowest level of the list.

- `modify_depth(x,0,fun)` is equivalent to `x[] <-fun(x)`.
- `modify_depth(x,1,fun)` is equivalent to `x <-modify(x,fun)`
- `modify_depth(x,2,fun)` is equivalent to `x <-modify(x,~ modify(.,fun))`

`.ragged` If `TRUE`, will apply to leaves, even if they're not at depth `.depth`. If `FALSE`, will throw an error if there are no elements at depth `.depth`.

Details

Since the transformation can alter the structure of the input; it's your responsibility to ensure that the transformation produces a valid output. For example, if you're modifying a data frame, `.f` must preserve the length of the input.

Value

An object the same class as `.x`

Genericity

`modify()` and variants are generic over classes that implement `length()`, `[[` and [[<- methods. If the default implementation is not compatible for your class, you can override them with your own methods.

If you implement your own `modify()` method, make sure it satisfies the following invariants:

```
modify(x, identity) === x
modify(x, compose(f, g)) === modify(x, g) %>% modify(f)
```

These invariants are known as the functor laws in computer science.

See Also

Other map variants: imap(), invoke(), lmap(), map2(), map_if(), map()

Examples

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

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

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

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

# Use modify2() to map over two vectors and preserve the type of
# the first one:
x <- c(foo = 1L, bar = 2L)
y <- c(TRUE, FALSE)
modify2(x, y, ~ if (.y) .x else 0L)

# Use a predicate function to decide whether to map a function:
modify_if(iris, is.factor, as.character)

# Specify an alternative with the `.else` argument:
modify_if(iris, is.factor, as.character, .else = as.integer)


# Modify at specified depth --------------------------
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 = 12:14, param2 = 15:17)
  )
)

# 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 %>% modify_depth(3, sum) %>% str()

# Note that vectorised operations will yield the same result when
# applied at the list level as when applied at the atomic result.
# The former is more efficient because it takes advantage of
# vectorisation.
l1 %>% modify_depth(3, `+`, 100L)
```

```
l1 %>% modify_depth(4, `+`, 100L)

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

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

# modify_depth() can be with other purrr functions to make them operate at
# a lower level. Here we ask pmap() to map paste() simultaneously over all
# elements of the objects at the second level. paste() is effectively
# mapped at level 3.
l1 %>% modify_depth(2, ~ pmap(., paste, sep = " / ")) %>% str()
```

`modify_in` *Modify a pluck location*

Description

- `assign_in()` takes a data structure and a pluck location, assigns a value there, and returns the modified data structure.
- `modify_in()` applies a function to a pluck location, assigns the result back to that location with `assign_in()`, and returns the modified data structure.

The pluck location must exist.

Usage

```
modify_in(.x, .where, .f, ...)

assign_in(x, where, value)
```

Arguments

`.x`	A vector or environment
`.where, where`	A pluck location, as a numeric vector of positions, a character vector of names, or a list combining both. The location must exist in the data structure.
`.f`	A function to apply at the pluck location given by `.where`.
`...`	Arguments passed to `.f`.
`x`	A vector or environment
`value`	A value to replace in `.x` at the pluck location.

See Also

`pluck()`

Examples

```
# Recall that pluck() returns a component of a data structure that
# might be arbitrarily deep
x <- list(list(bar = 1, foo = 2))
pluck(x, 1, "foo")

# Use assign_in() to modify the pluck location:
assign_in(x, list(1, "foo"), 100)

# modify_in() applies a function to that location and update the
# element in place:
modify_in(x, list(1, "foo"), ~ .x * 200)

# Additional arguments are passed to the function in the ordinary way:
modify_in(x, list(1, "foo"), `+`, 100)
```

---

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

```
negate("x")
negate(is.null)
negate(~ .x > 0)

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

These objects are imported from other packages. Follow the links below to see their documentation.

rlang %||%

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.

Note that an argument can only be partialised once.

Usage

```
partial(.f, ..., .env = NULL, .lazy = NULL, .first = NULL)
```

Arguments

.f	a function. For the output source to read well, this should be a named function.
...	named arguments to .f that should be partially applied. Pass an empty ... = argument to specify the position of future arguments relative to partialised ones. See rlang::call_modify() to learn more about this syntax. These dots support quasiquotation and quosures. If you unquote a value, it is evaluated only once at function creation time. Otherwise, it is evaluated each time the function is called.
.env	Soft-deprecated as of purrr 0.3.0. The environments are now captured via quosures.
.lazy	Soft-deprecated as of purrr 0.3.0. Please unquote the arguments that should be evaluated once at function creation time.
.first	Soft-deprecated as of purrr 0.3.0. Please pass an empty argument ... = to specify the position of future arguments.

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)
```

```
# 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")

# Note that you currently can't partialise arguments multiple times:
my_mean <- partial(mean, na.rm = TRUE)
my_mean <- partial(my_mean, na.rm = FALSE)
try(my_mean(1:10))


# The evaluation of arguments normally occurs "lazily". Concretely,
# this means that arguments are repeatedly evaluated across invocations:
f <- partial(runif, n = rpois(1, 5))
f
f()
f()

# You can unquote an argument to fix it to a particular value.
# Unquoted arguments are evaluated only once when the function is created:
f <- partial(runif, n = !!rpois(1, 5))
f
f()
f()


# By default, partialised arguments are passed before new ones:
my_list <- partial(list, 1, 2)
my_list("foo")

# Control the position of these arguments by passing an empty
# `... = ` argument:
my_list <- partial(list, 1, ... = , 2)
my_list("foo")
```

---

pluck *Pluck or chuck a single element from a vector or environment*

Description

pluck() and chuck() implement a generalised form of [[that allow you to index deeply and flexibly into data structures. pluck() consistently returns NULL when an element does not exist, chuck() always throws an error in that case.

Usage

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

chuck(.x, ...)

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

Arguments

`.x, x`	A vector or environment
`...`	A list of accessors for indexing into the object. Can be an integer position, a string name, or an accessor function (except for the assignment variants which only support names and positions). If the object being indexed is an S4 object, accessing it by name will return the corresponding slot. These dots support tidy dots features. In particular, if your accessors are stored in a list, you can splice that in with !!!.
`.default`	Value to use if target is empty or absent.
`value`	A value to replace in `.x` at the pluck location.

Details

- You can pluck or chuck with standard accessors like integer positions and string names, and also accepts arbitrary accessor functions, i.e. functions that take an object and return some internal piece.

  This is often more readable than a mix of operators and accessors because it reads linearly and is free of syntactic cruft. Compare: `accessor(x[[1]])$foo` to `pluck(x,1,accessor,"foo")`.
- These accessors never partial-match. This is unlike `$` which will select the `disp` object if you write `mtcars$di`.

See Also

`attr_getter()` for creating attribute getters suitable for use with `pluck()` and `chuck()`. `modify_in()` for applying a function to a pluck location.

Examples

```
# Let's create a list of data structures:
obj1 <- list("a", list(1, elt = "foo"))
obj2 <- list("b", list(2, elt = "bar"))
x <- list(obj1, obj2)


# pluck() provides a way of retrieving objects from such data
# structures using a combination of numeric positions, vector or
# list names, and accessor functions.

# Numeric positions index into the list by position, just like `[[`:
pluck(x, 1)
x[[1]]

pluck(x, 1, 2)
x[[1]][[2]]

# Supply names to index into named vectors:
pluck(x, 1, 2, "elt")
x[[1]][[2]][["elt"]]


# By default, pluck() consistently returns `NULL` when an element
# does not exist:
pluck(x, 10)
```

```
try(x[[10]])

# You can also supply a default value for non-existing elements:
pluck(x, 10, .default = NA)

# If you prefer to consistently fail for non-existing elements, use
# the opinionated variant chuck():
chuck(x, 1)
try(chuck(x, 10))
try(chuck(x, 1, 10))


# The map() functions use pluck() by default to retrieve multiple
# values from a list:
map(x, 2)

# Pass multiple indexes with a list:
map(x, list(2, "elt"))

# This is equivalent to:
map(x, pluck, 2, "elt")

# You can also supply a default:
map(x, list(2, "elt", 10), .default = "superb default")

# Or use the strict variant:
try(map(x, chuck, 2, "elt", 10))


# You can also assign a value in a pluck location with pluck<-:
pluck(x, 2, 2, "elt") <- "quuux"
x

# This is a shortcut for the prefix function assign_in():
y <- assign_in(x, list(2, 2, "elt"), value = "QUUUX")
y


# pluck() also supports accessor functions:
my_element <- function(x) x[[2]]$elt

# The accessor can then be passed to pluck:
pluck(x, 1, my_element)
pluck(x, 2, my_element)

# Even for this simple data structure, this is more readable than
# the alternative form because it requires you to read both from
# right-to-left and from left-to-right in different parts of the
# expression:
my_element(x[[1]])


# If you have a list of accessors, you can splice those in with `!!!`:
idx <- list(1, my_element)
pluck(x, !!!idx)
```

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 = NULL)
```

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. If NULL, values will be appended at the beginning even for x of length 0.

Value

A merged vector.

Examples

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

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

rate-helpers *Create delaying rate settings*

Description

These helpers create rate settings that you can pass to insistently(). You can also use them in your own functions with rate_sleep().

Usage

```
rate_delay(pause = 1, max_times = Inf)

rate_backoff(
  pause_base = 1,
  pause_cap = 60,
  pause_min = 1,
  max_times = 3,
```

```
  jitter = TRUE
)

is_rate(x)
```

Arguments

pause	Delay between attempts in seconds.
max_times	Maximum number of requests to attempt.
pause_base, pause_cap	rate_backoff() uses an exponential back-off so that each request waits pause_base * 2^i seconds, up to a maximum of pause_cap seconds.
pause_min	Minimum time to wait in the backoff; generally only necessary if you need pauses less than one second (which may not be kind to the server, use with caution!).
jitter	Whether to introduce a random jitter in the waiting time.
x	An object to test.

See Also

rate_sleep(), insistently()

Examples

```
# A delay rate waits the same amount of time:
rate <- rate_delay(0.02)
for (i in 1:3) rate_sleep(rate, quiet = FALSE)

# A backoff rate waits exponentially longer each time, with random
# jitter by default:
rate <- rate_backoff(pause_base = 0.2, pause_min = 0.005)
for (i in 1:3) rate_sleep(rate, quiet = FALSE)
```

---

rate_sleep *Wait for a given time*

Description

If the rate's internal counter exceeds the maximum number of times it is allowed to sleep, rate_sleep() throws an error of class purrr_error_rate_excess.

Usage

```
rate_sleep(rate, quiet = TRUE)

rate_reset(rate)
```

Arguments

rate	A rate object determining the waiting time.
quiet	If FALSE, prints a message displaying how long until the next request.

Details

Call `rate_reset()` to reset the internal rate counter to 0.

See Also

`rate_backoff()`, `insistently()`

---

rbernoulli *Generate random sample from a Bernoulli distribution*

Description

Generate random sample 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 sample from a discrete uniform distribution*

Description

Generate random sample 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))
table(rdunif(1e3, 10, -5))
```

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

Description

`reduce()` is an operation that combines the elements of a vector into a single value. The combination is driven by `.f`, a binary function that takes two values and returns a single value: reducing `f` over `1:3` computes the value `f(f(1,2),3)`.

Usage

```
reduce(.x, .f, ..., .init, .dir = c("forward", "backward"))

reduce2(.x, .y, .f, ..., .init)
```

Arguments

`.x`	A list or atomic vector.
`.f`	For `reduce()`, and `accumulate()`, a 2-argument function. The function will be passed the accumulated value as the first argument and the "next" value as the second argument. For `reduce2()` and `accumulate2()`, a 3-argument function. The function will be passed the accumulated value as the first argument, the next value of `.x` as the second argument, and the next value of `.y` as the third argument. The reduction terminates early if `.f` returns a value wrapped in a `done()`.
`...`	Additional arguments passed on to the mapped function.
`.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 a correct value when `.x` is empty. If missing, and `.x` is empty, will throw an error.
`.dir`	The direction of reduction as a string, one of `"forward"` (the default) or `"backward"`. See the section about direction below.
`.y`	For `reduce2()` and `accumulate2()`, an additional argument that is passed to `.f`. If `init` is not set, `.y` should be 1 element shorter than `.x`.

Direction

When `.f` is an associative operation like `+` or `c()`, the direction of reduction does not matter. For instance, reducing the vector `1:3` with the binary function `+` computes the sum `((1 + 2) + 3)` from the left, and the same sum `(1 + (2 + 3))` from the right.

In other cases, the direction has important consequences on the reduced value. For instance, reducing a vector with `list()` from the left produces a left-leaning nested list (or tree), while reducing `list()` from the right produces a right-leaning list.

Life cycle

`reduce_right()` is soft-deprecated as of purrr 0.3.0. Please use the `.dir` argument of `reduce()` instead. Note that the algorithm has changed. Whereas `reduce_right()` computed `f(f(3,2),1)`, reduce(.dir = \"backward\") computes `f(1,f(2,3))`. This is the standard way of reducing from the right.

To update your code with the same reduction as `reduce_right()`, simply reverse your vector and use a left reduction:

```
# Before:
reduce_right(1:3, f)

# After:
reduce(rev(1:3), f)
```

`reduce2_right()` is soft-deprecated as of purrr 0.3.0 without replacement. It is not clear what algorithmic properties should a right reduction have in this case. Please reach out if you know about a use case for a right reduction with a ternary function.

See Also

`accumulate()` for a version that returns all intermediate values of the reduction.

Examples

```
# Reducing `+` computes the sum of a vector while reducing `*`
# computes the product:
1:3 %>% reduce(`+`)
1:10 %>% reduce(`*`)

# When the operation is associative, the direction of reduction
# does not matter:
reduce(1:4, `+`)
reduce(1:4, `+`, .dir = "backward")

# However with non-associative operations, the reduced value will
# be different as a function of the direction. For instance,
# `list()` will create left-leaning lists when reducing from the
# right, and right-leaning lists otherwise:
str(reduce(1:4, list))
str(reduce(1:4, list, .dir = "backward"))

# reduce2() takes a ternary function and a second vector that is
# one element smaller than the first vector:
paste2 <- function(x, y, sep = ".") paste(x, y, sep = sep)
letters[1:4] %>% reduce(paste2)
letters[1:4] %>% reduce2(c("-", ".", "-"), paste2)

x <- list(c(0, 1), c(2, 3), c(4, 5))
y <- list(c(6, 7), c(8, 9))
reduce2(x, y, paste)


# You can shortcircuit a reduction and terminate it early by
# returning a value wrapped in a done(). In the following example
# we return early if the result-so-far, which is passed on the LHS,
# meets a condition:
paste3 <- function(out, input, sep = ".") {
  if (nchar(out) > 4) {
    return(done(out))
  }
  paste(out, input, sep = sep)
```

```
}
letters %>% reduce(paste3)

# Here the early return branch checks the incoming inputs passed on
# the RHS:
paste4 <- function(out, input, sep = ".") {
  if (input == "j") {
    return(done(out))
  }
  paste(out, input, sep = sep)
}
letters %>% reduce(paste4)
```

`rep_along` *Repeat a value with matching length*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

rlang `rep_along`

`rerun` *Re-run expressions multiple times.*

Description

Questioning

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`.

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)`.

Lifecycle

`rerun()` is in the questioning lifecycle stage because we are no longer convinced NSE functions are a good fit for purrr. Also, `rerun(n,x)` can just as easily be expressed as `map(1:n,~ x)` (with the added benefit of being passed the current index as argument to the lambda).

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 that instead of generating side effects through printed output, messages, warnings, and errors, they return enhanced 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)

auto_browse(.f)
```

Arguments

`.f`	A function, formula, or vector (not necessarily atomic). If a **function**, it is used as is. If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments: • For a single argument function, use `.` • For a two argument function, use `.x` and `.y` • For more arguments, use `..1`, `..2`, `..3` etc This syntax allows you to create very compact anonymous functions. If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`otherwise`	Default value to use when an error occurs.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Details

If you would like to include a function created with `safely`, `slowly`, or `insistently` in a package, see faq-adverbs-export.

Value

safely: wrapped function instead returns a list with components result and error. If an error occurred, error is an error object and result has a default value (otherwise). Else error is NULL.

quietly: wrapped function instead returns a list with components result, output, messages and warnings.

possibly: wrapped function uses a default value (otherwise) whenever an error occurs.

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_))

# For interactive usage, auto_browse() is useful because it automatically
# starts a browser() in the right place.
f <- function(x) {
  y <- 20
  if (x > 5) {
    stop("!")
  } else {
    x
  }
}
if (interactive()) {
  map(1:6, auto_browse(f))
}

# It doesn't make sense to use auto_browse with primitive functions,
# because they are implemented in C so there's no useful environment
# for you to interact with.
```

---

set_names *Set names in a vector*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

rlang set_names

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

Description

Questioning

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.

Value

A list.

Life cycle

`splice()` is in the questioning lifecycle stage as of purrr 0.3.0. We are now favouring the !!! syntax enabled by `rlang::list2()`.

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()
```

transpose *Transpose a list.*

Description

Transpose 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, .names = NULL)
```

Arguments

`.l`	A list of vectors to transpose. The first element is used as the template; you'll get a warning if a subsequent element has a different length.
`.names`	For efficiency, `transpose()` bases the return structure on the first component of `.l` by default. Specify `.names` to override this.

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.

Value

A list with indexing transposed compared to `.l`.

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()

# Provide explicit component names to prevent loss of those that don't
# appear in first component
ll <- list(
  list(x = 1, y = "one"),
```

```
  list(z = "deux", x = 2)
)
ll %>% transpose()
nms <- ll %>% map(names) %>% reduce(union)
ll %>% transpose(.names = nms)
```

---

vec_depth *Compute the depth of a vector*

Description

The depth of a vector is basically how many levels that you can index into it.

Usage

```
vec_depth(x)
```

Arguments

x	A vector

Value

An integer.

Examples

```
x <- list(
  list(),
  list(list()),
  list(list(list(1)))
)
vec_depth(x)
x %>% map_int(vec_depth)
```

---

zap *Zap an element*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

rlang zap

Index