Package ‘purrr’

July 23, 2025

Title Functional Programming Tools

Version 1.1.0

Description A complete and consistent functional programming toolkit for
R.

License MIT + file LICENSE

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

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

Depends R (>= 4.1)

Imports cli (>= 3.6.1), lifecycle (>= 1.0.3), magrittr (>= 1.5.0),
rlang (>= 1.1.1), vctrs (>= 0.6.3)

Suggests carrier (>= 0.2.0), covr, dplyr (>= 0.7.8), httr, knitr,
lubridate, mirai (>= 2.4.0), rmarkdown, testthat (>= 3.0.0),
tibble, tidyselect

LinkingTo cli

VignetteBuilder knitr

Biarch true

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate, tidyr

Config/testthat/edition 3

Config/testthat/parallel TRUE

Encoding UTF-8

RoxygenNote 7.3.2

NeedsCompilation yes

Author Hadley Wickham [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4757-117X>),
Lionel Henry [aut],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Hadley Wickham <hadley@posit.co>

Repository CRAN

Date/Publication 2025-07-10 17:50:02 UTC

Contents

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"),
  .simplify = NA,
  .ptype = NULL
)

accumulate2(.x, .y, .f, ..., .init, .simplify = NA, .ptype = NULL)
```

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. We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:

```
# Instead of
x |> map(f, 1, 2, collapse = ",")
# do:
x |> map(\(x) f(x, 1, 2, collapse = ","))
```

	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.
`.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.
`.simplify`	If `NA`, the default, the accumulated list of results is simplified to an atomic vector if possible. If `TRUE`, the result is simplified, erroring if not possible. If `FALSE`, the result is not simplified, always returning a list.
`.ptype`	If `simplify` is `NA` or `TRUE`, optionally supply a vector prototype to enforce the output type.
`.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 first element for a
# backward (left) accumulation, and in the last 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")

# By ignoring the input vector (nxt), you can turn output of one step into
# the input for the next. This code takes 10 steps of a random walk:
accumulate(1:10, \(acc, nxt) acc + rnorm(1), .init = 0)

# `accumulate2()` is a version of `accumulate()` that works with
# 3-argument functions and one additional vector:
paste2 <- function(acc, nxt, sep = ".") paste(acc, nxt, 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)

map(1:5, \(i) rnorm(100)) |>
  set_names(paste0("sim", 1:5)) |>
  map(\(l) accumulate(l, \(acc, nxt) .05 + acc + nxt)) |>
  map(\(x) tibble(value = x, step = 1:100)) |>
  list_rbind(names_to = "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. No longer recommended. 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(\(x) x + 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)
```

`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
```

`auto_browse` *Wrap a function so it will automatically* `browse()` *on error*

Description

A function wrapped with `auto_browse()` will automatically enter an interactive debugger using `browser()` when ever it encounters an error.

Usage

```
auto_browse(.f)
```

Arguments

`.f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `compose()`, `insistently()`, `negate()`, `partial()`, `possibly()`, `quietly()`, `safely()`, `slowly()`

Examples

```
# 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))
}
```

---

chuck — *Get an element deep within a nested data structure, failing if it doesn't exist*

Description

`chuck()` implements a generalised form of `[[` that allow you to index deeply and flexibly into data structures. If the index you are trying to access does not exist (or is `NULL`), it will throw (i.e. chuck) an error.

Usage

```
chuck(.x, ...)
```

Arguments

`.x`	A vector or environment
`...`	A list of accessors for indexing into the object. Can be an positive integer, a negative integer (to index from the right), a string (to index into names), 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. Dynamic dots are supported. In particular, if your accessors are stored in a list, you can splice that in with `!!!`.

See Also

`pluck()` for a quiet equivalent.

Examples

```
x <- list(a = 1, b = 2)

# When indexing an element that doesn't exist `[[` sometimes returns NULL:
x[["y"]]
# and sometimes errors:
try(x[[3]])

# chuck() consistently errors:
try(chuck(x, "y"))
try(chuck(x, 3))
```

`compose` *Compose multiple functions together to create a new function*

Description

Create a new function that is the composition of multiple functions, i.e. `compose(f, g)` is equivalent to `function(...) f(g(...))`.

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. Dynamic dots are supported. 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

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `insistently()`, `negate()`, `partial()`, `possibly()`, `quietly()`, `safely()`, `slowly()`

Examples

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

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

fn <- compose(\(x) paste(x, "foo"), \(x) paste(x, "bar"))
fn("input")

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

`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 vector.
`.f`	A function, specified in one of the following ways:

- A named function, e.g. `mean`.
- An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`.
- A formula, e.g. `~ .x + 1`. You must use `.x` to refer to the first argument. No longer recommended.
- A string, integer, or list, e.g. `"idx"`, `1`, or `list("idx", 1)` which are shorthand for `\(x) pluck(x, "idx")`, `\(x) pluck(x, 1)`, and `\(x) pluck(x, "idx", 1)` respectively. Optionally supply `.default` to set a default value if the indexed element is `NULL` or does not exist.

`...`	Additional arguments passed on to `.p`.
`.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`	**[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"))
```

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

Description

- `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.

Usage

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

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

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

Arguments

`.x`	A list or vector.
`.p`	A predicate function (i.e. a function that returns either `TRUE` or `FALSE`) specified in one of the following ways: • A named function, e.g. `is.character`. • An anonymous function, e.g. `\(x) all(x < 0)` or `function(x) all(x < 0)`. • A formula, e.g. `~ all(.x < 0)`. You must use `.x` to refer to the first argument). No longer recommended.
`...`	Additional arguments passed on to `.p`.

Value

A logical vector of length 1.

Examples

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

# Missing values are propagated:
some(list(NA, FALSE), identity)
```

```
# If you need to use these functions in a context where missing values are
# unsafe (e.g. in `if ()` conditions), make sure to use safe predicates:
if (some(list(NA, FALSE), rlang::is_true)) "foo" else "bar"
```

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. We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function: `# Instead of` `x |> map(f, 1, 2, collapse = ",")` `# do:` `x |> map(\(x) f(x, 1, 2, collapse = ","))` This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

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(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_vec(.x, .f, ...)

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

Arguments

`.x`	A list or atomic vector.
`.f`	A function, specified in one of the following ways: • A named function, e.g. `paste`. • An anonymous function, e.g. `\(x, idx) x + idx` or `function(x, idx) x + idx`. • A formula, e.g. `~ .x + .y`. You must use `.x` to refer to the current element and `.y` to refer to the current index. No longer recommended. **[Experimental]** Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

Value

A vector the same length as `.x`.

See Also

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

Examples

```
imap_chr(sample(10), paste)

imap_chr(sample(10), \(x, idx) paste0(idx, ": ", x))

iwalk(mtcars, \(x, idx) cat(idx, ": ", median(x), "\n", sep = ""))
```

insistently — *Transform a function to wait then retry after an error*

Description

`insistently()` takes a function and modifies it to retry after given amount of time whenever it errors.

Usage

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

Arguments

`f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.
`rate`	A rate object. Defaults to jittered exponential backoff.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

`httr::RETRY()` is a special case of `insistently()` for HTTP verbs.

Other adverbs: `auto_browse()`, `compose()`, `negate()`, `partial()`, `possibly()`, `quietly()`, `safely()`, `slowly()`

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)

# 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()
```

`in_parallel` *Parallelization in purrr*

Description

[Experimental]

All map functions allow parallelized operation using **mirai**.

Wrap functions passed to the `.f` argument of `map()` and its variants with `in_parallel()`.

`in_parallel()` is a **purrr** adverb that plays two roles:

- It is a signal to purrr verbs like `map()` to go ahead and perform computations in parallel.
- It helps you create self-contained functions that are isolated from your workspace. This is important because the function is packaged up (serialized) to be sent across to parallel processes. Isolation is critical for performance because it prevents accidentally sending very large objects between processes.

For maps to actually be performed in parallel, the user must also set `mirai::daemons()`, otherwise they fall back to sequential processing. `mirai::require_daemons()` may be used to enforce the use of parallel processing. See the section 'Daemons settings' below.

Usage

```
in_parallel(.f, ...)
```

Arguments

`.f`	A fresh formula or function. "Fresh" here means that they should be declared in the call to `in_parallel()`.
`...`	Named arguments to declare in the environment of the function.

Value

A 'crate' (classed function).

Creating self-contained functions

- They should call package functions with an explicit `::` namespace. For instance `ggplot()` from the ggplot2 package must be called with its namespace prefix: `ggplot2::ggplot()`. An alternative is to use `library()` within the function to attach a package to the search path, which allows subsequent use of package functions without the explicit namespace.
- They should declare any data they depend on. You can declare data by supplying additional named arguments to `...`. When supplying an anonymous function to a locally-defined function of the form `\(x) fun(x)`, the function `fun` itself must be supplied to `...`. The entire call would then be of the form: `in_parallel(\(x) fun(x), fun = fun)`.

`in_parallel()` is a simple wrapper of `carrier::crate()` and you may refer to that package for more details.

Example usage:

```
# The function needs to be freshly-defined, so instead of:
mtcars |> map_dbl(in_parallel(sum))
# Use an anonymous function:
mtcars |> map_dbl(in_parallel(\(x) sum(x)))

# Package functions need to be explicitly namespaced, so instead of:
map(1:3, in_parallel(\(x) vec_init(integer(), x)))
# Use :: to namespace all package functions:
map(1:3, in_parallel(\(x) vctrs::vec_init(integer(), x)))

fun <- function(x) { x + x %% 2 }
# Operating in parallel, locally-defined objects will not be found:
map(1:3, in_parallel(\(x) x + fun(x)))
# Use the ... argument to supply those objects:
map(1:3, in_parallel(\(x) x + fun(x), fun = fun))
```

When to use

Parallelizing a map using 'n' processes does not automatically lead to it taking 1/n of the time. Additional overhead from setting up the parallel task and communicating with parallel processes eats into this benefit, and can outweigh it for very short tasks or those involving large amounts of data. The threshold at which parallelization becomes clearly beneficial will differ according to your individual setup and task, but a rough guide would be in the order of 100 microseconds to 1 millisecond for each map iteration.

Daemons settings

How and where parallelization occurs is determined by `mirai::daemons()`. This is a function from the **mirai** package that sets up daemons (persistent background processes that receive parallel computations) on your local machine or across the network.

Daemons must be set prior to performing any parallel map operation, otherwise `in_parallel()` will fall back to sequential processing. To ensure that maps are always performed in parallel, put `mirai::require_daemons()` before the map.

It is usual to set daemons once per session. You can leave them running on your local machine as they consume almost no resources whilst waiting to receive tasks. The following sets up 6 daemons locally:

```
mirai::daemons(6)
```

Function arguments:

- `n`: the number of daemons to launch on your local machine, e.g. `mirai::daemons(6)`. As a rule of thumb, for maximum efficiency this should be (at most) one less than the number of cores on your machine, leaving one core for the main R process.
- `url` and `remote`: used to set up and launch daemons for distributed computing over the network. See `mirai::daemons()` documentation for more details.

Resetting daemons:

Daemons persist for the duration of your session. To reset and tear down any existing daemons:

```
mirai::daemons(0)
```

All daemons automatically terminate when your session ends. You do not need to explicitly terminate daemons in this instance, although it is still good practice to do so.

Note: it should always be for the user to set daemons. If you are using parallel map within a package, do not make any `mirai::daemons()` calls within the package, as it should always be up to the user how they wish to set up parallel processing e.g. using local or remote daemons. This also helps prevent inadvertently spawning too many daemons if functions are used recursively within each other.

References

purrr's parallelization is powered by **mirai**. See the mirai website for more details.

See Also

map() for usage examples.

Examples

```
# Run in interactive sessions only as spawns additional processes

slow_lm <- function(formula, data) {
  Sys.sleep(0.5)
  lm(formula, data)
}

# Example of a 'crate' returned by in_parallel(). The object print method
# shows the size of the crate and any objects contained within:
crate <- in_parallel(\(df) slow_lm(mpg ~ disp, data = df), slow_lm = slow_lm)
crate

# Use mirai::mirai() to test that a crate is self-contained
# by running it in a daemon and collecting its return value:
mirai::mirai(crate(mtcars), crate = crate) |> mirai::collect_mirai()
```

---

keep *Keep/discard elements based on their values*

Description

`keep()` selects all elements where `.p` evaluates to `TRUE`; `discard()` selects all elements where `.p` evaluates to `FALSE`. `compact()` discards elements where `.p` evaluates to an empty vector.

Usage

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

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

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

Arguments

`.x`	A list or vector.
`.p`	A predicate function (i.e. a function that returns either `TRUE` or `FALSE`) specified in one of the following ways: • A named function, e.g. `is.character`. • An anonymous function, e.g. `\(x) all(x < 0)` or `function(x) all(x < 0)`. • A formula, e.g. `~ all(.x < 0)`. You must use `.x` to refer to the first argument). No longer recommended.
`...`	Additional arguments passed on to `.p`.

Details

In other languages, `keep()` and `discard()` are often called `select()`/ `filter()` and `reject()`/ `drop()`, but those names are already taken in R. `keep()` is similar to `Filter()`, but the argument order is more convenient, and the evaluation of the predicate function `.p` is stricter.

See Also

`keep_at()`/`discard_at()` to keep/discard elements by name.

Examples

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

# Or use a formula
rep(10, 10) |>
  map(sample, 5) |>
  keep(\(x) 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()
```

`keep_at` *Keep/discard elements based on their name/position*

Description

Keep/discard elements based on their name/position

Usage

```
keep_at(x, at)

discard_at(x, at)
```

Arguments

`x`	A list or atomic vector.
`at`	A logical, integer, or character vector giving the elements to select. Alternatively, a function that takes a vector of names, and returns a logical, integer, or character vector of elements to select. **[Deprecated]**: if the tidyselect package is installed, you can use `vars()` and tidyselect helpers to select elements.

See Also

`keep()`/`discard()` to keep/discard elements by value.

Examples

```
x <- c(a = 1, b = 2, cat = 10, dog = 15, elephant = 5, e = 10)
x |> keep_at(letters)
x |> discard_at(letters)

# Can also use a function
x |> keep_at(\(x) nchar(x) == 3)
x |> discard_at(\(x) nchar(x) == 3)
```

`list_assign` *Modify a list*

Description

- `list_assign()` modifies the elements of a list by name or position.
- `list_modify()` modifies the elements of a list recursively.
- `list_merge()` merges the elements of a list recursively.

`list_modify()` is inspired by `utils::modifyList()`.

Usage

```
list_assign(.x, ..., .is_node = NULL)

list_modify(.x, ..., .is_node = NULL)

list_merge(.x, ..., .is_node = NULL)
```

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 by position. Dynamic dots are supported. In particular, if your replacement values are stored in a list, you can splice that in with `!!!`.
`.is_node`	A predicate function that determines whether an element is a node (by returning `TRUE`) or a leaf (by returning `FALSE`). The default value, `NULL`, treats simple lists as nodes and everything else (including richer objects like data frames and linear models) as leaves, using `vctrs::obj_is_list()`. To recurse into all objects built on lists use `is.list()`.

Examples

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

# Update values
str(list_assign(x, a = 1))

# Replace values
str(list_assign(x, z = 5))
str(list_assign(x, z = NULL))
str(list_assign(x, z = list(a = 1:5)))

# Replace recursively with list_modify(), leaving the other elements of z alone
str(list_modify(x, z = list(a = 1:5)))

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

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

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

`list_c` *Combine list elements into a single data structure*

Description

- `list_c()` combines elements into a vector by concatenating them together with `vctrs::vec_c()`.
- `list_rbind()` combines elements into a data frame by row-binding them together with `vctrs::vec_rbind()`.
- `list_cbind()` combines elements into a data frame by column-binding them together with `vctrs::vec_cbind()`.

Usage

```
list_c(x, ..., ptype = NULL)

list_cbind(
  x,
  ...,
  name_repair = c("unique", "universal", "check_unique"),
  size = NULL
)

list_rbind(x, ..., names_to = rlang::zap(), ptype = NULL)
```

Arguments

`x`	A list. For `list_rbind()` and `list_cbind()` the list must only contain only data frames or `NULL`.
`...`	These dots are for future extensions and must be empty.
`ptype`	An optional prototype to ensure that the output type is always the same.
`name_repair`	One of `"unique"`, `"universal"`, or `"check_unique"`. See `vctrs::vec_as_names()` for the meaning of these options.
`size`	An optional integer size to ensure that every input has the same size (i.e. number of rows).
`names_to`	By default, `names(x)` are lost. To keep them, supply a string to `names_to` and the names will be saved into a column with that name. If `names_to` is supplied and `x` is not named, the position of the elements will be used instead of the names.

Examples

```
x1 <- list(a = 1, b = 2, c = 3)
list_c(x1)

x2 <- list(
  a = data.frame(x = 1:2),
  b = data.frame(y = "a")
)
list_rbind(x2)
list_rbind(x2, names_to = "id")
list_rbind(unname(x2), names_to = "id")

list_cbind(x2)
```

list_flatten *Flatten a list*

Description

Flattening a list removes a single layer of internal hierarchy, i.e. it inlines elements that are lists leaving non-lists alone.

Usage

```
list_flatten(
  x,
  ...,
  name_spec = "{outer}_{inner}",
  name_repair = c("minimal", "unique", "check_unique", "universal")
)
```

Arguments

`x`	A list.
`...`	These dots are for future extensions and must be empty.
`name_spec`	If both inner and outer names are present, control how they are combined. Should be a glue specification that uses variables `inner` and `outer`.
`name_repair`	One of `"minimal"`, `"unique"`, `"universal"`, or `"check_unique"`. See `vctrs::vec_as_names()` for the meaning of these options.

Value

A list of the same type as `x`. The list might be shorter if `x` contains empty lists, the same length if it contains lists of length 1 or no sub-lists, or longer if it contains lists of length > 1.

Examples

```
x <- list(1, list(2, 3), list(4, list(5)))
x |> list_flatten() |> str()
x |> list_flatten() |> list_flatten() |> str()

# Flat lists are left as is
list(1, 2, 3, 4, 5) |> list_flatten() |> str()

# Empty lists will disappear
list(1, list(), 2, list(3)) |> list_flatten() |> str()

# Another way to see this is that it reduces the depth of the list
x <- list(
  list(),
  list(list())
)
```

```
x |> pluck_depth()
x |> list_flatten() |> pluck_depth()

# Use name_spec to control how inner and outer names are combined
x <- list(x = list(a = 1, b = 2), y = list(c = 1, d = 2))
x |> list_flatten() |> names()
x |> list_flatten(name_spec = "{outer}") |> names()
x |> list_flatten(name_spec = "{inner}") |> names()
```

`list_simplify` *Simplify a list to an atomic or S3 vector*

Description

Simplification maintains a one-to-one correspondence between the input and output, implying that each element of `x` must contain a one element vector or a one-row data frame. If you don't want to maintain this correspondence, then you probably want either `list_c()`/`list_rbind()` or `list_flatten()`.

Usage

```
list_simplify(x, ..., strict = TRUE, ptype = NULL)
```

Arguments

`x`	A list.
`...`	These dots are for future extensions and must be empty.
`strict`	What should happen if simplification fails? If `TRUE` (the default) it will error. If `FALSE` and `ptype` is not supplied, it will return `x` unchanged.
`ptype`	An optional prototype to ensure that the output type is always the same.

Value

A vector the same length as `x`.

Examples

```
list_simplify(list(1, 2, 3))

# Only works when vectors are length one and have compatible types:
try(list_simplify(list(1, 2, 1:3)))
try(list_simplify(list(1, 2, "x")))

# Unless you strict = FALSE, in which case you get the input back:
list_simplify(list(1, 2, 1:3), strict = FALSE)
list_simplify(list(1, 2, "x"), strict = FALSE)
```

list_transpose *Transpose a list*

Description

list_transpose() turns a list-of-lists "inside-out". For instance it turns a pair of lists into a list of pairs, or a list of pairs into a pair of lists. For example, if you had a list of length n where each component had values a and b, list_transpose() would make a list with elements a and b that contained lists of length n.

It's called transpose because x[["a"]][["b"]] is equivalent to list_transpose(x)[["b"]][["a"]], i.e. transposing a list flips the order of indices in a similar way to transposing a matrix.

Usage

```
list_transpose(
  x,
  ...,
  template = NULL,
  simplify = NA,
  ptype = NULL,
  default = NULL
)
```

Arguments

x	A list of vectors to transpose.
...	These dots are for future extensions and must be empty.
template	A "template" that describes the output list. Can either be a character vector (where elements are extracted by name), or an integer vector (where elements are extracted by position). Defaults to the union of the names of the elements of x, or if they're not present, the union of the integer indices.
simplify	Should the result be simplified? • TRUE: simplify or die trying. • NA: simplify if possible. • FALSE: never try to simplify, always leaving as a list. Alternatively, a named list specifying the simplification by output element.
ptype	An optional vector prototype used to control the simplification. Alternatively, a named list specifying the prototype by output element.
default	A default value to use if a value is absent or NULL. Alternatively, a named list specifying the default by output element.

Examples

```
# list_transpose() is useful in conjunction with safely()
x <- list("a", 1, 2)
y <- x |> map(safely(log))
y |> str()
# Put all the errors and results together
y |> list_transpose() |> str()
# Supply a default result to further simplify
y |> list_transpose(default = list(result = NA)) |> str()

# list_transpose() will try to simplify by default:
x <- list(list(a = 1, b = 2), list(a = 3, b = 4), list(a = 5, b = 6))
x |> list_transpose()
# this makes list_tranpose() not completely symmetric
x |> list_transpose() |> list_transpose()

# use simplify = FALSE to always return lists:
x |> list_transpose(simplify = FALSE) |> str()
x |>
  list_transpose(simplify = FALSE) |>
  list_transpose(simplify = FALSE) |> str()

# Provide an explicit template if you know which elements you want to extract
ll <- list(
  list(x = 1, y = "one"),
  list(z = "deux", x = 2)
)
ll |> list_transpose()
ll |> list_transpose(template = c("x", "y", "z"))
ll |> list_transpose(template = 1)

# And specify a default if you want to simplify
ll |> list_transpose(template = c("x", "y", "z"), default = NA)
```

`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()`, except instead of mapping over `.x[[i]]`, they instead map over `.x[i]`.

This has several advantages:

- It makes it possible to work with functions that exclusively take a list.
- It allows `.f` to access the attributes of the encapsulating list, like `names()`.
- It allows `.f` to return a larger or small list than it receives changing the size of the output.

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 a length-1 list and returns a list (of any length.)
`...`	Additional arguments passed on to the mapped function. We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function: `# Instead of` `x |> map(f, 1, 2, collapse = ",")` `# do:` `x |> map(\(x) f(x, 1, 2, collapse = ","))` This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.
`.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 logical, integer, or character vector giving the elements to select. Alternatively, a function that takes a vector of names, and returns a logical, integer, or character vector of elements to select. **[Deprecated]**: if the tidyselect package is installed, you can use `vars()` and tidyselect helpers to select elements.

Value

A list or data frame, matching `.x`. There are no guarantees about the length.

See Also

Other map variants: `imap()`, `map()`, `map2()`, `map_depth()`, `map_if()`, `modify()`, `pmap()`

Examples

```
set.seed(1014)

# Let's write a function that returns a larger list or an empty list
# depending on some condition. It also uses the input name to name the
# output
```

```
maybe_rep <- function(x) {
  n <- rpois(1, 2)
  set_names(rep_len(x, n), paste0(names(x), seq_len(n)))
}

# 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) |> str()

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

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

map — *Apply a function to each element of a 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). For these functions, `.f` must return a length-1 vector of the appropriate type.
- `map_vec()` simplifies to the common type of the output. It works with most types of simple vectors like Date, POSIXct, factors, etc.
- `walk()` calls `.f` for its side-effect and returns the input `.x`.

Usage

```
map(.x, .f, ..., .progress = FALSE)

map_lgl(.x, .f, ..., .progress = FALSE)

map_int(.x, .f, ..., .progress = FALSE)

map_dbl(.x, .f, ..., .progress = FALSE)

map_chr(.x, .f, ..., .progress = FALSE)

map_vec(.x, .f, ..., .ptype = NULL, .progress = FALSE)

walk(.x, .f, ..., .progress = FALSE)
```

Arguments

`.x` A list or atomic vector.

`.f` A function, specified in one of the following ways:

- A named function, e.g. `mean`.
- An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`.
- A formula, e.g. `~ .x + 1`. You must use `.x` to refer to the first argument. No longer recommended.
- A string, integer, or list, e.g. `"idx"`, `1`, or `list("idx", 1)` which are shorthand for `\(x) pluck(x, "idx")`, `\(x) pluck(x, 1)`, and `\(x) pluck(x, "idx", 1)` respectively. Optionally supply `.default` to set a default value if the indexed element is `NULL` or does not exist.

[Experimental]

Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

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

We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:

```
# Instead of
x |> map(f, 1, 2, collapse = ",")
# do:
x |> map(\(x) f(x, 1, 2, collapse = ","))
```

This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress` Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

`.ptype` If `NULL`, the default, the output type is the common type of the elements of the result. Otherwise, supply a "prototype" giving the desired type of output.

Value

The output length is determined by the length of the input. The output names are determined by the input names. The output type is determined by the suffix:

- No suffix: a list; `.f()` can return anything.
- `_lgl()`, `_int()`, `_dbl()`, `_chr()` return a logical, integer, double, or character vector respectively; `.f()` must return a compatible atomic vector of length 1.
- `_vec()` return an atomic or S3 vector, the same type that `.f` returns. `.f` can return pretty much any type of vector, as long as its length 1.
- `walk()` returns the input `.x` (invisibly). This makes it easy to use in a pipe. The return value of `.f()` is ignored.

Any errors thrown by `.f` will be wrapped in an error with class purrr_error_indexed.

See Also

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

Other map variants: imap(), lmap(), map2(), map_depth(), map_if(), modify(), pmap()

Examples

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

# You can also use an anonymous function
1:10 |>
  map(\(x) 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(\(food) paste(food, "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(mtcars$cyl) |>
  map(\(df) lm(mpg ~ wt, data = df)) |>
  map(summary) |>
  map_dbl("r.squared")


# Run in interactive sessions only as spawns additional processes

# To use parallelized map:
# 1. Set daemons (number of parallel processes) first:
mirai::daemons(2)

# 2. Wrap .f with in_parallel():
mtcars |> map_dbl(in_parallel(\(x) mean(x)))

# Note that functions from packages should be fully qualified with `pkg::`
# or call `library(pkg)` within the function
1:10 |>
  map(in_parallel(\(x) vctrs::vec_init(integer(), x))) |>
  map_int(in_parallel(\(x) { library(vctrs); vec_size(x) }))

# A locally-defined function (or any required variables)
# should be passed via ... of in_parallel():
slow_lm <- function(formula, data) {
  Sys.sleep(0.5)
  lm(formula, data)
}

mtcars |>
  split(mtcars$cyl) |>
  map(in_parallel(\(df) slow_lm(mpg ~ disp, data = df), slow_lm = slow_lm))

# Tear down daemons when no longer in use:
mirai::daemons(0)
```

map2 *Map over two inputs*

Description

These functions are variants of `map()` that iterate over two arguments at a time.

Usage

```
map2(.x, .y, .f, ..., .progress = FALSE)

map2_lgl(.x, .y, .f, ..., .progress = FALSE)
```

```
map2_int(.x, .y, .f, ..., .progress = FALSE)

map2_dbl(.x, .y, .f, ..., .progress = FALSE)

map2_chr(.x, .y, .f, ..., .progress = FALSE)

map2_vec(.x, .y, .f, ..., .ptype = NULL, .progress = FALSE)

walk2(.x, .y, .f, ..., .progress = FALSE)
```

Arguments

`.x, .y` A pair of vectors, usually the same length. If not, a vector of length 1 will be recycled to the length of the other.

`.f` A function, specified in one of the following ways:

- A named function.
- An anonymous function, e.g. `\(x, y) x + y` or `function(x, y) x + y`.
- A formula, e.g. `~ .x + .y`. You must use `.x` to refer to the current element of `x` and `.y` to refer to the current element of `y`. No longer recommended.

[Experimental]

Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

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

We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:

```
# Instead of
x |> map(f, 1, 2, collapse = ",")
# do:
x |> map(\(x) f(x, 1, 2, collapse = ","))
```

This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress` Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

`.ptype` If `NULL`, the default, the output type is the common type of the elements of the result. Otherwise, supply a "prototype" giving the desired type of output.

Value

The output length is determined by the length of the input. The output names are determined by the input names. The output type is determined by the suffix:

- No suffix: a list; `.f()` can return anything.
- `_lgl()`, `_int()`, `_dbl()`, `_chr()` return a logical, integer, double, or character vector respectively; `.f()` must return a compatible atomic vector of length 1.

- `_vec()` return an atomic or S3 vector, the same type that `.f` returns. `.f` can return pretty much any type of vector, as long as its length 1.
- `walk()` returns the input `.x` (invisibly). This makes it easy to use in a pipe. The return value of `.f()` is ignored.

Any errors thrown by `.f` will be wrapped in an error with class purrr_error_indexed.

See Also

Other map variants: `imap()`, `lmap()`, `map()`, `map_depth()`, `map_if()`, `modify()`, `pmap()`

Examples

```
x <- list(1, 1, 1)
y <- list(10, 20, 30)

map2(x, y, \(x, y) x + y)
# Or just
map2(x, y, `+`)

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

`map_depth` *Map/modify elements at given depth*

Description

`map_depth()` calls `map(.y, .f)` on all `.y` at the specified `.depth` in `.x`. `modify_depth()` calls `modify(.y, .f)` on `.y` at the specified `.depth` in `.x`.

Usage

```
map_depth(.x, .depth, .f, ..., .ragged = .depth < 0, .is_node = NULL)

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

Arguments

`.x` A list or atomic vector.

`.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, \(y) map(y, fun))`

`.f` A function, specified in one of the following ways:

- A named function, e.g. `mean`.
- An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`.
- A formula, e.g. `~ .x + 1`. You must use `.x` to refer to the first argument. No longer recommended.
- A string, integer, or list, e.g. `"idx"`, `1`, or `list("idx", 1)` which are shorthand for `\(x) pluck(x, "idx")`, `\(x) pluck(x, 1)`, and `\(x) pluck(x, "idx", 1)` respectively. Optionally supply `.default` to set a default value if the indexed element is `NULL` or does not exist.

[Experimental]

Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

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

We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:

```
# Instead of
x |> map(f, 1, 2, collapse = ",")
# do:
x |> map(\(x) f(x, 1, 2, collapse = ","))
```

This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.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`.

`.is_node` A predicate function that determines whether an element is a node (by returning `TRUE`) or a leaf (by returning `FALSE`). The default value, `NULL`, treats simple lists as nodes and everything else (including richer objects like data frames and linear models) as leaves, using `vctrs::obj_is_list()`. To recurse into all objects built on lists use `is.list()`.

See Also

`modify_tree()` for a recursive version of `modify_depth()` that allows you to apply a function to every leaf or every node.

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

Other modify variants: `modify()`, `modify_tree()`

Examples

```
# map_depth() -------------------------------------------------
# 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))
x |> str()
x |> map_depth(2, \(y) paste(y, collapse = "/")) |> str()
```

```
# Equivalent to:
x |> map(\(y) map(y, \(z) paste(z, collapse = "/"))) |> str()

# When ragged is TRUE, `.f()` will also be passed leaves at depth < `.depth`
x <- list(1, list(1, list(1, list(1, 1))))
x |> str()
x |> map_depth(4, \(x) length(unlist(x)), .ragged = TRUE) |> str()
x |> map_depth(3, \(x) length(unlist(x)), .ragged = TRUE) |> str()
x |> map_depth(2, \(x) length(unlist(x)), .ragged = TRUE) |> str()
x |> map_depth(1, \(x) length(unlist(x)), .ragged = TRUE) |> str()
x |> map_depth(0, \(x) length(unlist(x)), .ragged = TRUE) |> str()

# modify_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()

# 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, \(x) pmap(x, paste, sep = " / ")) |> str()
```

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`.

Usage

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

map_at(.x, .at, .f, ..., .progress = 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, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. You must use `.x` to refer to the first argument. No longer recommended. • A string, integer, or list, e.g. `"idx"`, `1`, or `list("idx", 1)` which are shorthand for `\(x) pluck(x, "idx")`, `\(x) pluck(x, 1)`, and `\(x) pluck(x, "idx", 1)` respectively. Optionally supply `.default` to set a default value if the indexed element is `NULL` or does not exist. **[Experimental]** Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.else`	A function applied to elements of `.x` for which `.p` returns `FALSE`.
`.at`	A logical, integer, or character vector giving the elements to select. Alternatively, a function that takes a vector of names, and returns a logical, integer, or character vector of elements to select. **[Deprecated]**: if the tidyselect package is installed, you can use `vars()` and tidyselect helpers to select elements.

`.progress`	Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.

See Also

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

Examples

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

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

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

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

`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_in()` modifies a single element in a `pluck()` location.

Usage

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

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

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

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

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

Arguments

`.x`	A vector.
`.f`	A function specified in the same way as the corresponding map function.

`...`	Additional arguments passed on to the mapped function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> map(\(x) f(x, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.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 logical, integer, or character vector giving the elements to select. Alternatively, a function that takes a vector of names, and returns a logical, integer, or character vector of elements to select. **[Deprecated]**: if the tidyselect package is installed, you can use `vars()` and tidyselect helpers to select elements.
`.y`	A vector, usually the same length as `.x`.

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()`, `lmap()`, `map()`, `map2()`, `map_depth()`, `map_if()`, `pmap()`

Other modify variants: `map_depth()`, `modify_tree()`

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 = sample(c(TRUE, FALSE), 100, replace = TRUE), y = 1:100) |>
  list_transpose(simplify = FALSE) |>
  modify_if("x", \(l) list(x = l$x, y = l$y * 100)) |>
  list_transpose()

# 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, \(x, cond) if (cond) 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_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.

Usage

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

assign_in(x, where, value)
```

Arguments

`.x, 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`.
`value`	A value to replace in `.x` at the pluck location. Use `zap()` to instead remove the element.

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:
str(assign_in(x, list(1, "foo"), 100))
# Or zap to remove it
str(assign_in(x, list(1, "foo"), zap()))

# Like pluck(), this works even when the element (or its parents) don't exist
pluck(x, 1, "baz")
str(assign_in(x, list(2, "baz"), 100))

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

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

`modify_tree` *Recursively modify a list*

Description

`modify_tree()` allows you to recursively modify a list, supplying functions that either modify each leaf or each node (or both).

Usage

```
modify_tree(
  x,
  ...,
  leaf = identity,
  is_node = NULL,
  pre = identity,
  post = identity
)
```

Arguments

`x`	A list.
`...`	Reserved for future use. Must be empty
`leaf`	A function applied to each leaf.
`is_node`	A predicate function that determines whether an element is a node (by returning `TRUE`) or a leaf (by returning `FALSE`). The default value, `NULL`, treats simple lists as nodes and everything else (including richer objects like data frames and linear models) as leaves, using `vctrs::obj_is_list()`. To recurse into all objects built on lists use `is.list()`.
`pre, post`	Functions applied to each node. `pre` is applied on the way "down", i.e. before the leaves are transformed with `leaf`, while `post` is applied on the way "up", i.e. after the leaves are transformed.

See Also

Other modify variants: `map_depth()`, `modify()`

Examples

```
x <- list(list(a = 2:1, c = list(b1 = 2), b = list(c2 = 3, c1 = 4)))
x |> str()

# Transform each leaf
x |> modify_tree(leaf = \(x) x + 100) |>  str()

# Recursively sort the nodes
sort_named <- function(x) {
  nms <- names(x)
  if (!is.null(nms)) {
    x[order(nms)]
  } else {
    x
   }
}
x |> modify_tree(post = sort_named) |> str()
```

negate — *Negate a predicate function so it selects what it previously rejected*

Description

Negating a function changes `TRUE` to `FALSE` and `FALSE` to `TRUE`.

Usage

```
negate(.p)
```

Arguments

`.p`	A predicate function (i.e. a function that returns either `TRUE` or `FALSE`) specified in one of the following ways: • A named function, e.g. `is.character`. • An anonymous function, e.g. `\(x) all(x < 0)` or `function(x) all(x < 0)`. • A formula, e.g. `~ all(.x < 0)`. You must use `.x` to refer to the first argument). No longer recommended.

Value

A new predicate function.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `partial()`, `possibly()`, `quietly()`, `safely()`, `slowly()`

Examples

```
x <- list(x = 1:10, y = rbernoulli(10), z = letters)
x |> keep(is.numeric) |> names()
x |> keep(negate(is.numeric)) |> names()
# Same as
x |> discard(is.numeric)
```

partial — *Partially 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 = deprecated(),
  .lazy = deprecated(),
  .first = deprecated()
)
```

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. 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`	**[Deprecated]** The environments are now captured via quosures.
`.lazy`	**[Deprecated]** Please unquote the arguments that should be evaluated once at function creation time with `!!`.
`.first`	**[Deprecated]** Please pass an empty argument `... =` to specify the position of future arguments.

Details

`partial()` creates a function that takes `...` arguments. Unlike `compose()` and other function operators like `negate()`, it doesn't reuse the function signature of `.f`. This is because `partial()` explicitly supports NSE functions that use `substitute()` on their arguments. The only way to support those is to forward arguments through dots.

Other unsupported patterns:

- It is not possible to call `partial()` repeatedly on the same argument to pre-fill it with a different expression.
- It is not possible to refer to other arguments in pre-filled argument.

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `negate()`, `possibly()`, `quietly()`, `safely()`, `slowly()`

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 *Safely get or set an element deep within a nested data structure*

Description

`pluck()` implements a generalised form of `[[` that allow you to index deeply and flexibly into data structures. It always succeeds, returning `.default` if the index you are trying to access does not exist or is `NULL`.
`pluck<-()` is the assignment equivalent, allowing you to modify an object deep within a nested data structure.

`pluck_exists()` tells you whether or not an object exists using the same rules as pluck (i.e. a `NULL` element is equivalent to an absent element).

Usage

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

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

pluck_exists(.x, ...)
```

Arguments

`.x, x`	A vector or environment
`...`	A list of accessors for indexing into the object. Can be an positive integer, a negative integer (to index from the right), a string (to index into names), 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. Dynamic dots are supported. In particular, if your accessors are stored in a list, you can splice that in with `!!!`.
`.default`	Value to use if target is `NULL` or absent.
`value`	A value to replace in `.x` at the pluck location. Use `zap()` to instead remove the element.

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)
# same as x[[1]]

# Index from the back
pluck(x, -1)
# same as x[[2]]

pluck(x, 1, 2)
# same as x[[1]][[2]]

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

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

```
map_chr(x, 1)
map_int(x, c(2, 1))

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

pluck_depth *Compute the depth of a vector*

Description

The depth of a vector is how many levels that you can index/pluck into it. `pluck_depth()` was previously called `vec_depth()`.

Usage

```
pluck_depth(x, is_node = NULL)
```

Arguments

`x`	A vector
`is_node`	Optionally override the default criteria for determine an element can be recursed within. The default matches the behaviour of `pluck()` which can recurse into lists and expressions.

Value

An integer.

Examples

```
x <- list(
  list(),
  list(list()),
  list(list(list(1)))
)
pluck_depth(x)
x |> map_int(pluck_depth)
```

pmap *Map over multiple input simultaneously (in "parallel")*

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, i.e. they share the same notion of "parallel" as `base::pmax()` and `base::pmin()`.

Usage

```
pmap(.l, .f, ..., .progress = FALSE)

pmap_lgl(.l, .f, ..., .progress = FALSE)

pmap_int(.l, .f, ..., .progress = FALSE)

pmap_dbl(.l, .f, ..., .progress = FALSE)

pmap_chr(.l, .f, ..., .progress = FALSE)

pmap_vec(.l, .f, ..., .ptype = NULL, .progress = FALSE)

pwalk(.l, .f, ..., .progress = FALSE)
```

Arguments

`.l` A list of vectors. The length of `.l` determines the number of arguments that `.f` will be called with. Arguments will be supply by position if unnamed, and by name if named.

Vectors of length 1 will be recycled to any length; all other elements must be have the same length.

A data frame is an important special case of `.l`. It will cause `.f` to be called once for each row.

`.f` A function, specified in one of the following ways:

- A named function.
- An anonymous function, e.g. `\(x, y, z) x + y / z` or `function(x, y, z) x + y / z`
- A formula, e.g. `~ ..1 + ..2 / ..3`. No longer recommended.

[Experimental]

Wrap a function with `in_parallel()` to declare that it should be performed in parallel. See `in_parallel()` for more details. Use of `...` is not permitted in this context.

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

We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:

```
# Instead of
x |> map(f, 1, 2, collapse = ",")
# do:
x |> map(\(x) f(x, 1, 2, collapse = ","))
```

This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.progress`	Whether to show a progress bar. Use `TRUE` to turn on a basic progress bar, use a string to give it a name, or see progress_bars for more details.
`.ptype`	If `NULL`, the default, the output type is the common type of the elements of the result. Otherwise, supply a "prototype" giving the desired type of output.

Value

The output length is determined by the maximum length of all elements of `.l`. The output names are determined by the names of the first element of `.l`. The output type is determined by the suffix:

- No suffix: a list; `.f()` can return anything.
- `_lgl()`, `_int()`, `_dbl()`, `_chr()` return a logical, integer, double, or character vector respectively; `.f()` must return a compatible atomic vector of length 1.
- `_vec()` return an atomic or S3 vector, the same type that `.f` returns. `.f` can return pretty much any type of vector, as long as it is length 1.
- `pwalk()` returns the input `.l` (invisibly). This makes it easy to use in a pipe. The return value of `.f()` is ignored.

Any errors thrown by `.f` will be wrapped in an error with class purrr_error_indexed.

See Also

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

Examples

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

# 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)
```

possibly — *Wrap a function to return a value instead of an error*

Description

Create a modified version of `.f` that return a default value (`otherwise`) whenever an error occurs.

Usage

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

Arguments

`.f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`.

	• A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.
`otherwise`	Default value to use when an error occurs.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `negate()`, `partial()`, `quietly()`, `safely()`, `slowly()`

Examples

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

# The default, NULL, will be discarded with `list_c()`
list("a", 10, 100) |>
  map(possibly(log)) |>
  list_c()
```

`progress_bars` *Progress bars in purrr*

Description

purrr's map functions have a `.progress` argument that you can use to create a progress bar. `.progress` can be:

- `FALSE`, the default: does not create a progress bar.
- `TRUE`: creates a basic unnamed progress bar.
- A string: creates a basic progress bar with the given name.
- A named list of progress bar parameters, as described below.

It's good practice to name your progress bars, to make it clear what calculation or process they belong to. We recommend keeping the names under 20 characters, so the whole progress bar fits comfortably even on on narrower displays.

Progress bar parameters:

- `clear`: whether to remove the progress bar from the screen after termination. Defaults to `TRUE`.
- `format`: format string. This overrides the default format string of the progress bar type. It must be given for the `custom` type. Format strings may contain R expressions to evaluate in braces. They support cli pluralization, and styling and they can contain special progress variables.
- `format_done`: format string for successful termination. By default the same as `format`.
- `format_failed`: format string for unsuccessful termination. By default the same as `format`.
- `name`: progress bar name. This is by default the empty string and it is displayed at the beginning of the progress bar.
- `show_after`: numeric scalar. Only show the progress bar after this number of seconds. It overrides the `cli.progress_show_after` global option.
- `type`: progress bar type. Currently supported types are:
  - `iterator`: the default, a for loop or a mapping function,
  - `tasks`: a (typically small) number of tasks,
  - `download`: download of one file,
  - `custom`: custom type, `format` must not be `NULL` for this type. The default display is different for each progress bar type.

Further documentation:

purrr's progress bars are powered by cli, so see Introduction to progress bars in cli and Advanced cli progress bars for more details.

`quietly` *Wrap a function to capture side-effects*

Description

Create a modified version of `.f` that captures side-effects along with the return value of the function and returns a list containing the `result`, `output`, `messages` and `warnings`.

Usage

```
quietly(.f)
```

Arguments

`.f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `negate()`, `partial()`, `possibly()`, `safely()`, `slowly()`

Examples

```
f <- function() {
  print("Hi!")
  message("Hello")
  warning("How are ya?")
  "Gidday"
}
f()

f_quiet <- quietly(f)
str(f_quiet())
```

`rate-helpers` *Create delaying rate settings*

Description

These helpers create rate settings that you can pass to `insistently()` and `slowly()`. 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.

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)
```

`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()`, 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()`, 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 reduce function.
	We now generally recommend against using `...` to pass additional (constant) arguments to `.f`. Instead use a shorthand anonymous function:
	`# Instead of`
	`x |> reduce(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> reduce(\(x, y) f(x, y, 1, 2, collapse = ","))`
	This makes it easier to understand which arguments belong to which function and will tend to yield better error messages.

`.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()` 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.

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(`*`)

# By ignoring the input vector (nxt), you can turn output of one step into
# the input for the next. This code takes 10 steps of a random walk:
reduce(1:10, \(acc, nxt) acc + rnorm(1), .init = 0)

# 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)
```

`safely` *Wrap a function to capture errors*

Description

Creates a modified version of `.f` that always succeeds. It returns a list with components `result` and `error`. If the function succeeds, `result` contains the returned value and `error` is `NULL`. If an error occurred, `error` is an `error` object and `result` is either `NULL` or `otherwise`.

Usage

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

Arguments

`.f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.
`otherwise`	Default value to use when an error occurs.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `negate()`, `partial()`, `possibly()`, `quietly()`, `slowly()`

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()
```

slowly *Wrap a function to wait between executions*

Description

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

Usage

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

Arguments

`f`	A function to modify, specified in one of the following ways: • A named function, e.g. `mean`. • An anonymous function, e.g. `\(x) x + 1` or `function(x) x + 1`. • A formula, e.g. `~ .x + 1`. Only recommended if you require backward compatibility with older versions of R.
`rate`	A rate object. Defaults to a constant delay.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?

Value

A function that takes the same arguments as `.f`, but returns a different value, as described above.

Adverbs

This function is called an adverb because it modifies the effect of a function (a verb). If you'd like to include a function created an adverb in a package, be sure to read faq-adverbs-export.

See Also

Other adverbs: `auto_browse()`, `compose()`, `insistently()`, `negate()`, `partial()`, `possibly()`, `quietly()`, `safely()`

Examples

```
# For these example, we first create a custom rate
# 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(\(x) runif(1), rate = rate, quiet = FALSE)
out <- map(1:5, slow_runif)
```

Index