Package 'furrr'

May 8, 2026

Title Apply Mapping Functions in Parallel using Futures

Version 0.4.0

Description Implementations of the family of map() functions from 'purrr' that can be resolved using any 'future'-supported backend, e.g. parallel on the local machine or distributed on a compute cluster.

License MIT + file LICENSE

URL https://github.com/futureverse/furrr, https://furrr.futureverse.org/

BugReports https://github.com/futureverse/furrr/issues

Depends future (>= 1.70.0), R (>= 4.1.0)

Imports globals (>= 0.19.1), purrr (>= 1.2.1), rlang (>= 1.1.7), vctrs (>= 0.7.0)

Suggests carrier, covr, dplyr (>= 1.1.4), knitr, parallelly (>= 1.46.1), testthat (>= 3.3.2), tidyselect

Config/Needs/website progressr

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

Author Davis Vaughan [aut, cre] (ORCID: <https://orcid.org/0000-0003-4777-038X>), Henrik Bengtsson [aut] (ORCID: <https://orcid.org/0000-0002-7579-5165>), Matt Dancho [aut], Posit Software, PBC [cph, fnd]

Maintainer Davis Vaughan <davis@posit.co>

Repository CRAN

Date/Publication 2026-03-31 14:50:02 UTC

Contents

`furrr_options` *Options to fine tune furrr*

Description

`furrr_options()` returns an object that can be supplied as the `.options` argument for furrr functions, such as `future_map()`. The options are either used by furrr directly, or are passed on to `future::future()`.

Usage

```
furrr_options(
  ...,
  stdout = TRUE,
  conditions = "condition",
  globals = TRUE,
  packages = NULL,
  seed = FALSE,
  scheduling = 1,
  chunk_size = NULL,
  prefix = NULL
)
```

Arguments

`...`	These dots are reserved for future extensibility and must be empty.
`stdout`	A logical. • If `TRUE`, standard output of the underlying futures is relayed as soon as possible. • If `FALSE`, output is silenced by sinking it to the null device.
`conditions`	A character string of conditions classes to be relayed. The default is to relay all conditions, including messages and warnings. Errors are always relayed. To not relay any conditions (besides errors), use `conditions = character()`. To selectively ignore specific classes, use `conditions = structure("condition", exclude = "message")`.

`globals` A logical, a character vector, a named list, or `NULL` for controlling how globals are handled. For details, see the `Global variables` section below.

`packages` A character vector, or `NULL`. If supplied, this specifies packages that are guaranteed to be attached in the R environment where the future is evaluated.

`seed` A logical, an integer of length `1` or `7`, a list of `length(.x)` with pre-generated random seeds, or `NULL`. For details, see the `Reproducible random number generation (RNG)` section below.

`scheduling` A single integer, logical, or `Inf`. This argument controls the average number of futures ("chunks") per worker.

- If `0`, then a single future is used to process all elements of `.x`.
- If `1` or `TRUE`, then one future per worker is used.
- If `2`, then each worker will process two futures (provided there are enough elements in `.x`).
- If `Inf` or `FALSE`, then one future per element of `.x` is used.

This argument is only used if `chunk_size` is `NULL`.

`chunk_size` A single integer, `Inf`, or `NULL`. This argument controls the average number of elements per future (`"chunk"`). If `Inf`, then all elements are processed in a single future. If `NULL`, then `scheduling` is used instead to determine how `.x` is chunked.

`prefix` A single character string, or `NULL`. If a character string, then each future is assigned a label as `{prefix}-{chunk-id}`. If `NULL`, no labels are used.

Global variables

`globals` controls how globals are identified, similar to the `globals` argument of `future::future()`. Since all function calls use the same set of globals, furrr gathers globals upfront (once), which is more efficient than if it was done for each future independently.

- If `TRUE` or `NULL`, then globals are automatically identified and gathered.
- If a character vector of names is specified, then those globals are gathered.
- If a named list, then those globals are used as is.
- In all cases, `.f` and any `...` arguments are automatically passed as globals to each future created, as they are always needed.

Reproducible random number generation (RNG)

Unless `seed = FALSE`, furrr functions are guaranteed to generate the exact same sequence of random numbers *given the same initial seed / RNG state* regardless of the type of futures and scheduling ("chunking") strategy.

Setting `seed = NULL` is equivalent to `seed = FALSE`, except that the `future.rng.onMisuse` option is not consulted to potentially monitor the future for faulty random number usage. See the `seed` argument of `future::future()` for more details.

RNG reproducibility is achieved by pre-generating the random seeds for all iterations (over `.x`) by using L'Ecuyer-CMRG RNG streams. In each iteration, these seeds are set before calling `.f(.x[[i]], ...)`. *Note, for large* `length(.x)` *this may introduce a large overhead.*

A fixed seed may be given as an integer vector, either as a full L'Ecuyer-CMRG RNG seed of length 7, or as a seed of length 1 that will be used to generate a full L'Ecuyer-CMRG seed.

If seed = TRUE, then .Random.seed is returned if it holds a L'Ecuyer-CMRG RNG seed, otherwise one is created randomly.

If seed = NA, a L'Ecuyer-CMRG RNG seed is randomly created.

If none of the function calls .f(.x[[i]], ...) use random number generation, then seed = FALSE may be used.

In addition to the above, it is possible to specify a pre-generated sequence of RNG seeds as a list such that length(seed) == length(.x) and where each element is an integer seed that can be assigned to .Random.seed. Use this alternative with caution. *Note that* as.list(seq_along(.x)) *is not a valid set of such* .Random.seed *values.*

In all cases but seed = FALSE, after a furrr function returns, the RNG state of the calling R process is guaranteed to be "forwarded one step" from the RNG state before the call. This is true regardless of the future strategy / scheduling used. This is done in order to guarantee that an R script calling future_map() multiple times should be numerically reproducible given the same initial seed.

Note that you cannot expect identical results between map() and future_map() when using a .f that calls functions that generate random numbers, even when calling set.seed() ahead of time. For one thing, the default random number generation algorithm used by R during sequential processing is Mersenne-Twister, different from the L'Ecuyer-CMRG seeds used by furrr. But even aligning the RNGkind() would not be enough. map() itself would have to change to use the same parallel compatible RNG strategy as future_map() (pre-generating the seeds, and setting them before each .f invocation). At the end of the day, you have to accept that the following will produce different sequences of random numbers, but both are statistically sound:

```
set.seed(42)
purrr::map(1:10, ~ rnorm(1))

set.seed(42)
furrr::future_map(1:10, ~ rnorm(1), .options = furrr_options(seed = TRUE))
```

But importantly, the furrr::future_map() example will always produce the same sequence of random numbers, regardless of the plan() you choose:

```
plan(sequential)
set.seed(42)
furrr::future_map(1:10, ~ rnorm(1), .options = furrr_options(seed = TRUE))

plan(multisession, workers = 2)
set.seed(42)
furrr::future_map(1:10, ~ rnorm(1), .options = furrr_options(seed = TRUE))

plan(cluster, workers = workers)
set.seed(42)
furrr::future_map(1:10, ~ rnorm(1), .options = furrr_options(seed = TRUE))
```

Examples

```
furrr_options()
```

future_imap *Apply a function to each element of a vector, and its index via futures*

Description

These functions work the same as `purrr::imap()` functions, but allow you to map in parallel.

Usage

```
future_imap(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_chr(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_dbl(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_int(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_lgl(
  .x,
```

```
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_vec(
  .x,
  .f,
  ...,
  .ptype = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_dfr(
  .x,
  .f,
  ...,
  .id = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_imap_dfc(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_iwalk(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)
```

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`. Use `.x` to refer to the current element and `.y` to refer to the current index. No longer recommended.

`...` 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 |> future_map(f, 1, 2, collapse = ",")
# do:
x |> future_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.

`.options` The `future` specific options to use with the workers. This must be the result from a call to `furrr_options()`.

`.env_globals` The environment to look for globals required by `.x` and `....` Globals required by `.f` are looked up in the function environment of `.f`.

`.progress` A single logical. Should a progress bar be displayed? Only works with multisession, multicore, and multiprocess futures. Note that if a multicore/multisession future falls back to sequential, then a progress bar will not be displayed.

Warning: The `.progress` argument will be deprecated and removed in a future version of furrr in favor of using the more robust progressr package.

`.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.

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

Only applies to `_dfr` variant.

Value

A vector the same length as .x.

Examples

```
plan(multisession, workers = 2)

future_imap_chr(sample(10), ~ paste0(.y, ": ", .x))
```

future_map *Apply a function to each element of a vector via futures*

Description

These functions work the same as `purrr::map()` and its variants, but allow you to map in parallel.

Usage

```
future_map(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_chr(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_dbl(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_int(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_lgl(
  .x,
```

```
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_vec(
  .x,
  .f,
  ...,
  .ptype = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_dfr(
  .x,
  .f,
  ...,
  .id = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_dfc(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_walk(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .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`. 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 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 |> future_map(f, 1, 2, collapse = ",")
# do:
x |> future_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.

`.options` The `future` specific options to use with the workers. This must be the result from a call to `furrr_options()`.

`.env_globals` The environment to look for globals required by `.x` and `....` Globals required by `.f` are looked up in the function environment of `.f`.

`.progress` A single logical. Should a progress bar be displayed? Only works with multisession, multicore, and multiprocess futures. Note that if a multicore/multisession future falls back to sequential, then a progress bar will not be displayed.

Warning: The `.progress` argument will be deprecated and removed in a future version of furrr in favor of using the more robust progressr package.

`.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.

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

Only applies to `_dfr` variant.

Value

All functions return a vector the same length as `.x`.

- `future_map()` returns a list
- `future_map_lgl()` a logical vector
- `future_map_int()` an integer vector
- `future_map_dbl()` a double vector
- `future_map_chr()` a character vector

The output of `.f` will be automatically typed upwards, e.g. logical -> integer -> double -> character.

Examples

```
plan(multisession, workers = 2)

1:10 |>
  future_map(rnorm, n = 10, .options = furrr_options(seed = 123)) |>
  future_map_dbl(mean)

# If each element of the output is a data frame, use
# `future_map_dfr()` to row-bind them together:
mtcars |>
  split(mtcars$cyl) |>
  future_map(~ lm(mpg ~ wt, data = .x)) |>
  future_map_dfr(~ as.data.frame(t(as.matrix(coef(.)))))


# You can be explicit about what gets exported to the workers.
# To see this, use multisession (not multicore as the forked workers
# still have access to this environment)
plan(multisession)
x <- 1
y <- 2

# This will fail, y is not exported (no black magic occurs)
try(future_map(1, ~y, .options = furrr_options(globals = "x")))

# y is exported
future_map(1, ~y, .options = furrr_options(globals = "y"))
```

`future_map2` *Map over multiple inputs simultaneously via futures*

Description

These functions work the same as `purrr::map2()` and its variants, but allow you to map in parallel. Note that "parallel" as described in purrr is just saying that you are working with multiple inputs, and parallel in this case means that you can work on multiple inputs and process them all in parallel as well.

Usage

```
future_map2(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
```

```
  .progress = FALSE
)

future_map2_chr(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_dbl(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_int(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_lgl(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_vec(
  .x,
  .y,
  .f,
  ...,
```

```
  .ptype = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_dfr(
  .x,
  .y,
  .f,
  ...,
  .id = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map2_dfc(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap(
  .l,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_chr(
  .l,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_dbl(
  .l,
  .f,
```

```
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_int(
  .l,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_lgl(
  .l,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_vec(
  .l,
  .f,
  ...,
  .ptype = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_dfr(
  .l,
  .f,
  ...,
  .id = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pmap_dfc(
  .l,
  .f,
  ...,
```

```
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_walk2(
  .x,
  .y,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_pwalk(
  .l,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .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`. Use `.x` to refer to the current element of `x` and `.y` to refer to the current element of `y`. No longer recommended.

`...`	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 |> future_map(f, 1, 2, collapse = ",")`
	`# do:`
	`x |> future_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.

`.options`	The `future` specific options to use with the workers. This must be the result from a call to `furrr_options()`.
`.env_globals`	The environment to look for globals required by `.x` and `....` Globals required by `.f` are looked up in the function environment of `.f`.

`.progress` A single logical. Should a progress bar be displayed? Only works with multisession, multicore, and multiprocess futures. Note that if a multicore/multisession future falls back to sequential, then a progress bar will not be displayed.

Warning: The `.progress` argument will be deprecated and removed in a future version of furrr in favor of using the more robust progressr package.

`.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.

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

Only applies to `_dfr` variant.

`.l` A list of vectors. 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.

Value

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

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

Examples

```
plan(multisession, workers = 2)

x <- list(1, 10, 100)
y <- list(1, 2, 3)
z <- list(5, 50, 500)

future_map2(x, y, ~ .x + .y)

# Split into pieces, fit model to each piece, then predict
by_cyl <- split(mtcars, mtcars$cyl)
mods <- future_map(by_cyl, ~ lm(mpg ~ wt, data = .))
future_map2(mods, by_cyl, predict)

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

# Matching arguments by position
future_pmap(list(x, y, z), function(a, b ,c) a / (b + c))

# Vectorizing a function over multiple arguments
df <- data.frame(
  x = c("apple", "banana", "cherry"),
```

```
  pattern = c("p", "n", "h"),
  replacement = c("x", "f", "q"),
  stringsAsFactors = FALSE
)

future_pmap(df, gsub)
future_pmap_chr(df, gsub)
```

`future_map_if` *Apply a function to each element of a vector conditionally via futures*

Description

These functions work the same as `purrr::map_if()` and `purrr::map_at()`, but allow you to run them in parallel.

Usage

```
future_map_if(
  .x,
  .p,
  .f,
  ...,
  .else = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_map_at(
  .x,
  .at,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .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`. 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 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 |> future_map(f, 1, 2, collapse = ",")
# do:
x |> future_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`.

`.options` The `future` specific options to use with the workers. This must be the result from a call to `furrr_options()`.

`.env_globals` The environment to look for globals required by `.x` and `....` Globals required by `.f` are looked up in the function environment of `.f`.

`.progress` A single logical. Should a progress bar be displayed? Only works with multisession, multicore, and multiprocess futures. Note that if a multicore/multisession future falls back to sequential, then a progress bar will not be displayed.

Warning: The `.progress` argument will be deprecated and removed in a future version of furrr in favor of using the more robust progressr package.

`.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

Both functions return a list the same length as `.x` with the elements conditionally transformed.

Examples

```
plan(multisession, workers = 2)

# Modify the even elements
future_map_if(1:5, ~.x %% 2 == 0L, ~ -1)

future_map_at(1:5, c(1, 5), ~ -1)
```

future_modify *Modify elements selectively via futures*

Description

These functions work the same as purrr::modify() functions, but allow you to modify in parallel.

Usage

```
future_modify(
  .x,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_modify_at(
  .x,
  .at,
  .f,
  ...,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)

future_modify_if(
  .x,
  .p,
  .f,
  ...,
  .else = NULL,
  .options = furrr_options(),
  .env_globals = parent.frame(),
  .progress = FALSE
)
```

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 |> future_map(f, 1, 2, collapse = ",")
# do:
x |> future_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.

`.options` The `future` specific options to use with the workers. This must be the result from a call to `furrr_options()`.

`.env_globals` The environment to look for globals required by `.x` and `...`. Globals required by `.f` are looked up in the function environment of `.f`.

`.progress` A single logical. Should a progress bar be displayed? Only works with multisession, multicore, and multiprocess futures. Note that if a multicore/multisession future falls back to sequential, then a progress bar will not be displayed.

Warning: The `.progress` argument will be deprecated and removed in a future version of furrr in favor of using the more robust progressr package.

`.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.

`.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`.

Details

From purrr:

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`

Examples

```
plan(multisession, workers = 2)

# Convert each col to character, in parallel
future_modify(mtcars, as.character)

iris |>
 future_modify_if(is.factor, as.character) |>
 str()
```

```
mtcars |>
  future_modify_at(c(1, 4, 5), as.character) |>
  str()
```

Index