Package 'carrier'

October 28, 2026

Title Isolate Functions for Remote Execution

Version 0.4.0

Description Sending functions to remote processes can be wasteful of resources because they carry their environments with them. With the carrier package, it is easy to create functions that are isolated from their environment. These isolated functions, also called crates, print at the console with their total size and can be easily tested locally before being sent to a remote.

License MIT + file LICENSE

URL https://github.com/r-lib/carrier

BugReports https://github.com/r-lib/carrier/issues

Depends R (>= 4.1)

Imports lobstr, rlang (>= 1.0.1)

Suggests testthat (>= 3.0.0)

ByteCompile true

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.0

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-06-16

Encoding UTF-8

NeedsCompilation no

Author Lionel Henry [aut, cre],
Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Lionel Henry <lionel@posit.co>

Repository CRAN

Date/Publication 2026-10-28 10:40:02 UTC

Contents

crate 2
is_crate 4

Index **5**

`crate` *Crate a function to share with another process*

Description

`crate()` creates functions in a self-contained environment (technically, a child of the base environment). This has two advantages:

- They can easily be executed in another process.
- Their effects are reproducible. You can run them locally with the same results as on a different process.

Creating self-contained functions requires some care, see section below.

Usage

```
crate(
  .fn,
  ...,
  .parent_env = baseenv(),
  .error_arg = ".fn",
  .error_call = environment()
)
```

Arguments

`.fn` A formula or function. The function may be defined inside the call to `crate()` or already exist. For an existing function:

- Plain closures have their enclosing environment replaced with the crate's evaluation environment, so any data they depend on must be declared via `...`. This means closures that depend on captured state, such as those created by function factories (e.g. the result of `stats::ecdf()`), lose that state and fail when called. See Examples for how to crate such functions.
- Namespace functions and primitives are wrapped in a thin closure that delegates to them, leaving their original environment untouched.
- Existing crates are passed through unchanged.

Formulas are converted to purrr-like lambda functions using `rlang::as_function()`.

`...` Named arguments to declare in the environment of `.fn`. Declared data is only reachable from plain closures; it is an error to supply `...` when `.fn` is an existing crate, a namespace function, or a primitive.

`.parent_env` The default of `baseenv()` ensures that the evaluation environment of the crate is isolated from the search path. Specifying another environment such as the global environment allows this condition to be relaxed (but at the expense of no longer being able to rely on a local run giving the same results as one in a different process).

`.error_arg` An argument name as a string. This argument will be mentioned in error messages as the input that is at the origin of a problem.

`.error_call` The execution environment of a currently running function, e.g. `caller_env()`. The function will be mentioned in error messages as the source of the error. See the `call` argument of `abort()` for more information.

Creating self-contained functions

- They should call package functions with an explicit `::` namespace. This includes packages in the default search path with the exception of the base package. For instance `var()` from the stats package must be called with its namespace prefix: `stats::var(x)`.
- They should declare any data they depend on. You can declare data by supplying additional arguments or by unquoting objects with `!!`.
- A function (closure) supplied as an additional argument is itself crated. This means that it must be self-contained by supplying all objects required by it as further additional arguments, if not already supplied. Only functions directly supplied to `...` are crated, and containers such as lists are not recursively walked to find functions. Note that if an existing crate is supplied, this is not re-crated.

Examples

```
# You can create functions using the ordinary notation:
crate(\(x) stats::var(x))

# Or the formula notation:
crate(~ stats::var(.x))

# Declare data by supplying named arguments. You can test you have
# declared all necessary data by calling your crated function:
na_rm <- TRUE
fn <- crate(~ stats::var(.x, na.rm = na_rm))
try(fn(1:10))

# Declare `na_rm` by passing it as a named argument:
fn <- crate(~ stats::var(.x, na.rm = na_rm), na_rm = na_rm)
fn(1:10)

# The size of each declared object is shown in the crate printout:
fn


# You can also pass an already defined function. Its enclosing
# environment is replaced with the crate's evaluation environment,
# so any data it depends on must be declared via `...`:
fn <- function(x) toupper(x)
```

```
crate(fn)

# For the same reason, existing functions that depend on state
# captured in their environment, such as those created by function
# factories, lose that state and fail when called:
fn <- stats::ecdf(1:10)
crated <- crate(fn)
try(crated(5))

# Instead, construct the function within the crate:
crated <- crate(\(x) stats::ecdf(1:10)(x))
crated(5)
```

`is_crate` *Is an object a crate?*

Description

Is an object a crate?

Usage

```
is_crate(x)
```

Arguments

`x`	An object to test.

Index

abort(), *3*

crate, 2

is_crate, 4

rlang::as_function(), *2*

stats::ecdf(), *2*