Package 'carrier'

July 22, 2025

Title Isolate Functions for Remote Execution

Version 0.2.0

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 (>= 3.4.0)

Imports lobstr, rlang (>= 1.0.1)

Suggests covr, testthat (>= 3.0.0)

ByteCompile true

Config/testthat/edition 3

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

Encoding UTF-8

RoxygenNote 7.3.2

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 2025-06-23 08:00:02 UTC

Contents

`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 fresh formula or function. "Fresh" here means that they should be declared in the call to `crate()`. See examples if you need to crate a function that is already defined. Formulas are converted to purrr-like lambda functions using `rlang::as_function()`.
`...`	Named arguments to declare in the environment of `.fn`.
`.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 `!!`.

Examples

```
# You can create functions using the ordinary notation:
crate(function(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))

# For small data it is handy to unquote instead. Unquoting inlines
# objects inside the function. This is less verbose if your
# function depends on many small objects:
fn <- crate(~ stats::var(.x, na.rm = !!na_rm))
fn(1:10)

# One downside is that the individual sizes of unquoted objects
# won't be shown in the crate printout:
fn


# The function or formula you pass to crate() should defined inside
# the crate() call, i.e. you can't pass an already defined
# function:
fn <- function(x) toupper(x)
try(crate(fn))

# If you really need to crate an existing function, you can
# explicitly set its environment to the crate environment with the
# set_env() function from rlang:
crate(rlang::set_env(fn))
```

`is_crate` *Is an object a crate?*

Description

Is an object a crate?

Usage

```
is_crate(x)
```

Arguments

x	An object to test.

Index