Package 'lifecycle'

Title Manage the Life Cycle of your Package Functions

Version 1.0.5

Description Manage the life cycle of your exported functions with shared
conventions, documentation badges, and user-friendly deprecation
warnings.

License MIT + file LICENSE

URL https://lifecycle.r-lib.org/, https://github.com/r-lib/lifecycle

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

Depends R (>= 3.6)

Imports cli (>= 3.4.0), rlang (>= 1.1.0)

Suggests covr, knitr, lintr (>= 3.1.0), rmarkdown, testthat (>=
3.0.1), tibble, tidyverse, tools, vctrs, withr, xml2

VignetteBuilder knitr

Config/Needs/website tidyverse/tidytemplate, usethis

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation no

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

Maintainer Lionel Henry <lionel@posit.co>

Repository CRAN

Date/Publication 2026-01-08 08:00:02 UTC

Contents

Index   **12**

---

badge   *Embed a lifecycle badge in documentation*

Description

To include lifecycle badges in your documentation:

1. Call `usethis::use_lifecycle()` to copy the badge images into the `man/` folder of your package.

2. Call `lifecycle::badge()` inside R backticks to insert a lifecycle badge:

   ```
   #' `r lifecycle::badge("experimental")`
   #' `r lifecycle::badge("deprecated")`
   #' `r lifecycle::badge("superseded")`
   ```

   If the deprecated feature is a function, a good place for this badge is at the top of the topic description. If it is an argument, you can put the badge in the argument description.

The badge is displayed as an image in the HTML version of the documentation and as text otherwise.

`lifecycle::badge()` is run by roxygen at build time so you don't need to add lifecycle to `Imports:` just to use the badges. However, it's still good practice to add to `Suggests:` so that it will be available to package developers.

Usage

```
badge(stage)
```

Arguments

stage   A lifecycle stage as a string. Must be one of `"experimental"`, `"stable"`,
                `"superseded"`, or `"deprecated"`.

Value

An `Rd` expression describing the lifecycle stage.

Badges

- **[Experimental]** `lifecycle::badge("experimental")`
- **[Stable]** `lifecycle::badge("stable")`
- **[Superseded]** `lifecycle::badge("superseded")`
- **[Deprecated]** `lifecycle::badge("deprecated")`

The meaning of these stages is described in `vignette("stages")`.

---

deprecated *Mark an argument as deprecated*

Description

Signal deprecated argument by using self-documenting sentinel `deprecated()` as default argument. Test whether the caller has supplied the argument with `is_present()`.

Usage

```
deprecated()

is_present(arg)
```

Arguments

arg   A `deprecated()` function argument.

Magical defaults

We recommend importing `lifecycle::deprecated()` in your namespace and use it without the namespace qualifier.

In general, we advise against such magical defaults, i.e. defaults that cannot be evaluated by the user. In the case of `deprecated()`, the trade-off is worth it because the meaning of this default is obvious and there is no reason for the user to call `deprecated()` themselves.

Examples

```
foobar_adder <- function(foo, bar, baz = deprecated()) {
  # Check if user has supplied `baz` instead of `bar`
  if (lifecycle::is_present(baz)) {

    # Signal the deprecation to the user
    deprecate_warn("1.0.0", "foo::bar_adder(baz = )", "foo::bar_adder(bar = )")

    # Deal with the deprecated argument for compatibility
    bar <- baz
  }
```

```
  foo + bar
}

foobar_adder(1, 2)
foobar_adder(1, baz = 2)
```

---

deprecate_soft   *Deprecate functions and arguments*

Description

These functions provide three levels of verbosity for deprecated functions. Learn how to use them
in vignette("communicate").

- deprecate_soft() warns only if the deprecated function is called directly, i.e. a user is
  calling a function they wrote in the global environment or a developer is calling it in their
  package. It does not warn when called indirectly, i.e. the deprecation comes from code that
  you don't control.
- deprecate_warn() warns unconditionally.
- deprecate_stop() fails unconditionally.

Warnings are only issued once per session to avoid overwhelming the user. Control with options(lifecycle_verbosity).

Usage

```
deprecate_soft(
  when,
  what,
  with = NULL,
  details = NULL,
  id = NULL,
  env = caller_env(),
  user_env = caller_env(2)
)

deprecate_warn(
  when,
  what,
  with = NULL,
  details = NULL,
  id = NULL,
  always = FALSE,
  env = caller_env(),
  user_env = caller_env(2)
)

deprecate_stop(when, what, with = NULL, details = NULL, env = caller_env())
```

Arguments

when	A string giving the version when the behaviour was deprecated.
what	A string describing what is deprecated:

- Deprecate a whole function with `"foo()"`.
- Deprecate an argument with `"foo(arg)"`.
- Partially deprecate an argument with `"foo(arg = 'must be a scalar integer')"`.
- Deprecate anything else with a custom message by wrapping it in `I()`.

You can optionally supply the namespace: `"ns::foo()"`, but this is usually not needed as it will be inferred from the caller environment.

with	An optional string giving a recommended replacement for the deprecated behaviour. This takes the same form as `what`.
details	In most cases the deprecation message can be automatically generated from `with`. When it can't, use `details` to provide a hand-written message.

`details` can either be a single string or a character vector, which will be converted to a [bulleted list](). By default, info bullets are used. Provide a named vectors to override.

id	The id of the deprecation. A warning is issued only once for each id. Defaults to the generated message, but you should provide a unique `id` when the message in `details` is built programmatically and depends on inputs, or when you'd like to deprecate multiple functions but warn only once for all of them. Repeated calls to `deprecate_soft()` and `deprecate_warn()` are also much faster if you supply an `id` because it avoids spending time generating the message only to immediately exit if the once per session warning has already been thrown before.
env, user_env	Pair of environments that define where `deprecate_*()` was called (used to determine the package name) and where the function called the deprecating function was called (used to determine if `deprecate_soft()` should message).

These are only needed if you're calling `deprecate_*()` from an internal helper, in which case you should forward `env = caller_env()` and `user_env = caller_env(2)`.

always	If `FALSE`, the default, will warn once per session. If `TRUE`, will always warn in direct usages. Indirect usages keep warning once per session to avoid disrupting users who can't fix the issue. Only use `always = TRUE` after at least one release with the default.

Value

`NULL`, invisibly.

Conditions

- Deprecation warnings have class `lifecycle_warning_deprecated`.
- Deprecation errors have class `lifecycle_error_deprecated`.

See Also

[lifecycle()]()

Examples

```
# A deprecated function `foo`:
deprecate_warn("1.0.0", "foo()")

# A deprecated argument `arg`:
deprecate_warn("1.0.0", "foo(arg)")

# A partially deprecated argument `arg`:
deprecate_warn("1.0.0", "foo(arg = 'must be a scalar integer')")

# A deprecated function with a function replacement:
deprecate_warn("1.0.0", "foo()", "bar()")

# A deprecated function with a function replacement from a
# different package:
deprecate_warn("1.0.0", "foo()", "otherpackage::bar()")

# A deprecated function with custom message:
deprecate_warn(
  when = "1.0.0",
  what = "foo()",
  details = "Please use `otherpackage::bar(foo = TRUE)` instead"
)

# A deprecated function with custom bulleted list:
deprecate_warn(
  when = "1.0.0",
  what = "foo()",
  details = c(
    x = "This is dangerous",
    i = "Did you mean `safe_foo()` instead?"
  )
)
```

---

expect_deprecated   *Does expression produce lifecycle warnings or errors?*

Description

These functions are equivalent to [testthat::expect_warning()](#) and [testthat::expect_error()](#) but check specifically for lifecycle warnings or errors.

To test whether a deprecated feature still works without causing a deprecation warning, set the `lifecycle_verbosity` option to "quiet".

```
test_that("feature still works", {
  withr::local_options(lifecycle_verbosity = "quiet")
  expect_true(my_deprecated_function())
})
```

Usage

```
expect_deprecated(expr, regexp = NULL, ...)

expect_defunct(expr)
```

Arguments

expr	Expression that should produce a lifecycle warning or error.
regexp	Optional regular expression matched against the expected warning message.
...	Arguments passed on to [expect_match](#)
	fixed If TRUE, treats regexp as a string to be matched exactly (not a regular expressions). Overrides perl.
	perl logical. Should Perl-compatible regexps be used?

Details

expect_deprecated() sets the [lifecycle_verbosity](#) option to "warning" to enforce deprecation warnings which are otherwise only shown once per session.

```
last_lifecycle_warnings
```
Display last deprecation warnings

Description

last_lifecycle_warnings() returns a list of all warnings that occurred during the last top-level R command, along with a backtrace.

Use print(last_lifecycle_warnings(), simplify = level) to control the verbosity of the backtrace. The simplify argument supports one of "branch" (the default), "collapse", and "none" (in increasing order of verbosity).

Usage

```
last_lifecycle_warnings()
```

Examples

```
# These examples are not run because `last_lifecycle_warnings()` does not
# work well within knitr and pkgdown
## Not run:

f <- function() invisible(g())
g <- function() list(h(), i())
h <- function() deprecate_warn("1.0.0", "this()")
i <- function() deprecate_warn("1.0.0", "that()")
f()
```

```
# Print all the warnings that occurred during the last command:
last_lifecycle_warnings()


# By default, the backtraces are printed in their simplified form.
# Use `simplify` to control the verbosity:
print(last_lifecycle_warnings(), simplify = "none")

## End(Not run)
```

---

pkg_lifecycle_statuses

Lint usages of functions that have a non-stable life cycle.

Description

- lifecycle_linter() creates a linter for lifecycle annotations which can be included in a .lintr configuration if lintr is used directly.
- lint_lifecycle() dynamically queries the package documentation for packages in packages for lifecycle annotations and then searches the directory in path for usages of those functions.
- lint_tidyverse_lifecycle() is a convenience function to call lint_lifecycle() for all the packages in the tidyverse.
- pkg_lifecycle_statuses() returns a data frame of functions with lifecycle annotations for an installed package.

Usage

```
pkg_lifecycle_statuses(
  package,
  which = c("superseded", "deprecated", "questioning", "defunct", "experimental",
    "soft-deprecated", "retired")
)

lint_lifecycle(
  packages,
  path = ".",
  pattern = "(?i)[.](r|rmd|qmd|rnw|rhtml|rrst|rtex|rtxt)$",
  which = c("superseded", "deprecated", "questioning", "defunct", "experimental",
    "soft-deprecated", "retired"),
  symbol_is_undesirable = FALSE
)

lint_tidyverse_lifecycle(
  path = ".",
  pattern = "(?i)[.](r|rmd|qmd|rnw|rhtml|rrst|rtex|rtxt)$",
  which = c("superseded", "deprecated", "questioning", "defunct", "experimental",
```

```
      "soft-deprecated", "retired"),
  symbol_is_undesirable = FALSE
)

lifecycle_linter(
  packages = tidyverse::tidyverse_packages(),
  which = c("superseded", "deprecated", "questioning", "defunct", "experimental",
      "soft-deprecated", "retired"),
  symbol_is_undesirable = FALSE
)
```

Arguments

package	The name of an installed package.
which	Vector of lifecycle statuses to lint.
packages	One or more installed packages to query for lifecycle statuses.
path	The directory path to the files you want to search.
pattern	Any files matching this pattern will be searched. The default searches any files ending in `.R` or `.Rmd`.
symbol_is_undesirable	
	Also lint symbol usages, e.g. `lapply(x, is_na)`?

Examples

```
lintr::lint(
  text = "is_na(x)",
  linters = lifecycle_linter(packages = "rlang")
)
lintr::lint(
  text = "lapply(x, is_na)",
  linters = lifecycle_linter(packages = "rlang",
  symbol_is_undesirable = TRUE)
)
```

---

signal_stage   *Signal other experimental or superseded features*

Description

[Experimental]

`signal_stage()` allows you to signal life cycle stages other than deprecation (for which you should use [deprecate_warn()](#) and friends). There is no behaviour associated with this signal, it is currently purely a way to express intent at the call site. In the future, we hope to replace this with a standardized call to `base::declare()`.

Usage

```
signal_stage(stage, what, with = NULL, env = deprecated())
```

Arguments

stage	Life cycle stage, either "experimental" or "superseded".
what	String describing what feature the stage applies too, using the same syntax as [deprecate_warn()]().
with	An optional string giving a recommended replacement for a superseded function.
env	**[Deprecated]**

Examples

```
foofy <- function(x, y, z) {
  signal_stage("experimental", "foofy()")
  x + y / z
}
foofy(1, 2, 3)
```

---

verbosity   *Control the verbosity of deprecation signals*

Description

There are 3 levels of verbosity for deprecated functions: silence, warning, and error. Since the lifecycle package avoids disruptive warnings, the default level of verbosity depends on the lifecycle stage of the deprecated function, on the context of the caller (global environment or testthat unit tests cause more warnings), and whether the warning was already issued (see the help for [deprecation functions]()).

You can control the level of verbosity with the global option lifecycle_verbosity. It can be set to:

- "quiet" to suppress all deprecation messages.

- "default" or NULL to warn once per session.

- "warning" to warn every time.

- "error" to error instead of warning.

Note that functions calling [deprecate_stop()]() invariably throw errors.

Examples

```
if (rlang::is_installed("testthat")) {
  library(testthat)

  mytool <- function() {
    deprecate_soft("1.0.0", "mytool()")
    10 * 10
  }

  # Forcing the verbosity level is useful for unit testing. You can
  # force errors to test that the function is indeed deprecated:
  test_that("mytool is deprecated", {
    rlang::local_options(lifecycle_verbosity = "error")
    expect_error(mytool(), class = "defunctError")
  })

  # Or you can enforce silence to safely test that the function
  # still works:
  test_that("mytool still works", {
    rlang::local_options(lifecycle_verbosity = "quiet")
    expect_equal(mytool(), 100)
  })
}
```

Index