Package 'lifecycle'

July 27, 2019

Title Manage the Life Cycle of your Package Functions

Version 0.1.0

Description Manage the life cycle of your exported functions with shared conventions, documentation badges, and non-invasive deprecation warnings. The 'lifecycle' package defines four development stages (experimental, maturing, stable, and questioning) and three deprecation stages (soft-deprecated, deprecated, and defunct). It makes it easy to insert badges corresponding to these stages in your documentation. Usage of deprecated functions are signalled with increasing levels of non-invasive verbosity.

License GPL-3

Encoding UTF-8

LazyData true

Depends R (>= 3.2)

Imports glue,
rlang (>= 0.4.0)

Suggests covr,
crayon,
knitr,
rmarkdown,
testthat (>= 2.1.0)

Roxygen list(markdown = TRUE)

RoxygenNote 6.1.1

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

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

VignetteBuilder knitr

R topics documented:

badge 2
deprecated 3
deprecate_soft 3
last_warnings 5
verbosity 6

Index **8**

badge *Embed a lifecycle badge in documentation*

Description

Call `usethis::use_lifecycle()` to import the badges in your package. Then use the `lifecycle` Rd macro to insert a lifecycle badges in your documentation, with the relevant lifecycle stage as argument:

```
\lifecycle{experimental}
\lifecycle{soft-deprecated}
```

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

If the deprecated feature is a function, a good place for this badge is at the top of the topic description (if the deprecated function is documented with other functions, it might be a good idea to extract it in its own documentation topic to prevent confusion). If it is an argument, you can put the badge in the argument description.

Usage

```
badge(stage)
```

Arguments

`stage`	A lifecycle stage as a string, one of: `"experimental"`, `"maturing"`, `"stable"`, `"questioning"`, `"archived"`, `"soft-deprecated"`, `"deprecated"`, `"defunct"`.

Details

The `lifecycle{}` macro is made available by adding this field to DESCRIPTION (this is done automatically by `usethis::use_lifecycle()`):

```
RdMacros: lifecycle
```

The macro expands to this expression:

```
\Sexpr[results=rd, stage=render]{lifecycle::badge("experimental")}
```

Value

An `Rd` expression describing the lifecycle stage.

Badges

- \lifecycle{experimental}: **Experimental**
- \lifecycle{maturing}: **Maturing**
- \lifecycle{stable}: **Stable**
- \lifecycle{questioning}: **Questioning**
- \lifecycle{archived}: **Archived**
- \lifecycle{soft-deprecated}: **Soft-deprecated**
- \lifecycle{deprecated}: **Deprecated**
- \lifecycle{defunct}: **Defunct**

deprecated *Mark an argument as deprecated*

Description

Signal deprecated argument by using self-documenting sentinel `deprecated()` as default argument. It returns `rlang::missing_arg()`, so you can test whether the user supplied the argument with `rlang::is_missing()` (see examples).

Usage

```
deprecated()
```

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 (!rlang::is_missing(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.

- `deprecate_soft()` warns only if the deprecated function is called from the global environment (so the user can change their script) or from the package currently being tested (so the package developer can fix the package). Use for soft-deprecated functions.

- `deprecate_warn()` warns unconditionally. Use for deprecated functions.
- `deprecate_stop()` fails unconditionally. Use for defunct functions.

Warnings are only issued once per session to avoid overwhelming the user. See the verbosity option to control this behaviour.

Usage

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

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

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

Arguments

`when`	The package version when function/argument was deprecated.
`what`	If the deprecated feature is a whole function, the function name: `"foo()"`. If it's an argument that is being deprecated, the function call should include the argument: `"foo(arg = )"`. You can optionally supply the namespace: `"ns::foo()"`. If not supplied, it is inferred from the caller environment.
`with`	An optional replacement for the deprecated feature. This should be a string of the same form as `what`.
`details`	The deprecation message is generated from `when`, `what`, and `with`. You can additionally supply a string `details` to be appended to the message.
`id`	The id of the deprecation. A warning is issued only once for each `id`. Defaults to the generated message, but you should give 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.
`env`	The environment in which the deprecated function was called. A warning is issued if called from the global environment. If testthat is running, a warning is also called if the deprecated function was called from the package being tested. This typically doesn't need to be specified, unless you call `deprecate_soft()` or `deprecate_warn()` from an internal helper. In that case, you need to forward the calling environment.

Value

`NULL`, invisibly.

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 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 an argument replacement:
deprecate_warn("1.0.0", "foo()", "foo(bar = )")
```

---

last_warnings *Display last deprecation warnings*

Description

Call these helpers to see the last deprecation warnings along with their backtrace:

- `last_warnings()` returns a list of all warnings that occurred during the last top-level R command.
- `last_warning()` returns only the last.

If you call these in the console, these warnings are printed with a backtrace. Pass the `simplify` argument to control the verbosity of the backtrace. It supports one of `"branch"` (the default), `"collapse"`, and `"none"` (in increasing order of verbosity).

Usage

```
last_warnings()

last_warning()
```

Examples

```
# These examples are not run because `last_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_warnings()
```

```
# Print only the last one:
last_warning()


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


## End(Not run)
```

`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:

- `"default"` or `NULL` for the default non-disruptive settings.
- `"quiet"`, `"warning"` or `"error"` to force silence, warnings or errors for deprecated functions.

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::with_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::with_options(lifecycle_verbosity = "quiet", {
      expect_equal(mytool(), 100)
    })
```

```
    })
  }
```

Index

badge, 2

deprecate_soft, 3
deprecate_stop *(*deprecate_soft*)*, 3
deprecate_stop(), *6*
deprecate_warn *(*deprecate_soft*)*, 3
deprecated, 3
deprecation functions, *6*

last_warning *(*last_warnings*)*, 5
last_warnings, 5
lifecycle(), *4*

rlang::is_missing(), *3*
rlang::missing_arg(), *3*

verbosity, 6
verbosity option, *4*