Package 'teal.code'

February 14, 2025

Type Package

Title Code Storage and Execution Class for 'teal' Applications

Version 0.6.1

Date 2025-02-13

Description Introduction of 'qenv' S4 class, that facilitates code execution and reproducibility in 'teal' applications.

License Apache License 2.0

URL https://insightsengineering.github.io/teal.code/, https://github.com/insightsengineering/teal.code

BugReports https://github.com/insightsengineering/teal.code/issues

Depends methods, R (>= 4.0)

Imports checkmate (>= 2.1.0), cli (>= 3.4.0), grDevices, lifecycle (>= 0.2.0), rlang (>= 1.1.0), stats, utils

Suggests knitr (>= 1.42), rmarkdown (>= 2.23), shiny (>= 1.6.0), testthat (>= 3.1.8), withr (>= 2.0.0)

VignetteBuilder knitr, rmarkdown

RdMacros lifecycle

Config/Needs/verdepcheck mllg/checkmate, r-lib/cli, r-lib/lifecycle, r-lib/rlang, r-lib/cli, yihui/knitr, rstudio/rmarkdown, rstudio/shiny, r-lib/testthat, r-lib/withr

Config/Needs/website insightsengineering/nesttemplate

Encoding UTF-8

Language en-US

RoxygenNote 7.3.2

Collate 'qenv-c.R' 'qenv-class.R' 'qenv-errors.R' 'qenv-concat.R' 'qenv-constructor.R' 'qenv-eval_code.R' 'qenv-extract.R' 'qenv-get_code.R' 'qenv-get_env.R' 'qenv-get_messages.r' 'qenv-get_var.R' 'qenv-get_warnings.R' 'qenv-join.R' 'qenv-length.R' 'qenv-show.R' 'qenv-within.R' 'teal.code-package.R' 'utils-get_code_dependency.R' 'utils.R'

NeedsCompilation no

Author Dawid Kaledkowski [aut, cre],
Aleksander Chlebowski [aut],
Marcin Kosinski [aut],
Pawel Rucki [aut],
Nikolas Burkoff [aut],
Mahmoud Hallal [aut],
Maciej Nasinski [aut],
Konrad Pagacz [aut],
Junlue Zhao [aut],
Chendi Liao [rev],
Dony Unardi [rev],
F. Hoffmann-La Roche AG [cph, fnd]

Maintainer Dawid Kaledkowski <dawid.kaledkowski@roche.com>

Repository CRAN

Date/Publication 2025-02-14 05:40:05 UTC

Contents

`c.qenv` *Join* `qenv` *objects*

Description

Checks and merges two `qenv` objects into one `qenv` object.

The `join()` function is superseded by the `c()` function.

Usage

```
## S3 method for class 'qenv'
c(...)

## S3 method for class 'qenv.error'
c(...)

join(x, y)
```

Arguments

`...`	(qenv or qenv.error).
`x`	(qenv)
`y`	(qenv)

Details

Any common code at the start of the qenvs is only placed once at the start of the joined qenv. This allows consistent behavior when joining qenvs which share a common ancestor. See below for an example.

There are some situations where `join()` cannot be properly performed, such as these three scenarios:

1. Both qenv objects contain an object of the same name but are not identical.
   Example:

   ```
   x <- eval_code(qenv(), expression(mtcars1 <- mtcars))
   y <- eval_code(qenv(), expression(mtcars1 <- mtcars['wt']))

   z <- c(x, y)
   # Error message will occur
   ```

   In this example, `mtcars1` object exists in both `x` and `y` objects but the content are not identical. `mtcars1` in the `x qenv` object has more columns than `mtcars1` in the `y qenv` object (only has one column).

2. `join()` will look for identical code elements in both qenv objects. The index position of these code elements must be the same to determine the evaluation order. Otherwise, `join()` will throw an error message.
   Example:

   ```
   common_q <- eval_code(qenv(), expression(v <- 1))
   x <- eval_code(
     common_q,
     "x <- v"
   )
   y <- eval_code(
     common_q,
     "y <- v"
   ```

```
)
z <- eval_code(
  y,
  "z <- v"
)
q <- c(x, y)
join_q <- c(q, z)
# Error message will occur

# Check the order of evaluation based on the id slot
```

The error occurs because the index position of common code elements in the two objects is not the same.

3. The usage of temporary variable in the code expression could cause `join()` to fail.
   Example:

```
common_q <- qenv()
x <- eval_code(
  common_q,
  "x <- numeric(0)
   for (i in 1:2) {
     x <- c(x, i)
   }"
)
y <- eval_code(
  common_q,
  "y <- numeric(0)
   for (i in 1:3) {
     y <- c(y, i)
   }"
)
q <- join(x,y)
# Error message will occur

# Check the value of temporary variable i in both objects
x$i # Output: 2
y$i # Output: 3
```

`c()` fails to provide a proper result because of the temporary variable `i` exists in both objects but has different value. To fix this, we can set `i <- NULL` in the code expression for both objects.

```
common_q <- qenv()
x <- eval_code(
  common_q,
  "x <- numeric(0)
   for (i in 1:2) {
     x <- c(x, i)
   }
   # dummy i variable to fix it
```

```
   i <- NULL"
)
y <- eval_code(
  common_q,
  "y <- numeric(0)
   for (i in 1:3) {
     y <- c(y, i)
   }
   # dummy i variable to fix it
   i <- NULL"
)
q <- c(x,y)
```

Value

qenv object.

Examples

```
q <- qenv()
q1 <- within(q, {
  iris1 <- iris
  mtcars1 <- mtcars
})
q1 <- within(q1, iris2 <- iris)
q2 <- within(q1, mtcars2 <- mtcars)
qq <- c(q1, q2)
cat(get_code(qq))

q <- qenv()
q1 <- eval_code(q, expression(iris1 <- iris, mtcars1 <- mtcars))
q2 <- q1
q1 <- eval_code(q1, "iris2 <- iris")
q2 <- eval_code(q2, "mtcars2 <- mtcars")
qq <- join(q1, q2)
cat(get_code(qq))

common_q <- eval_code(q, quote(x <- 1))
y_q <- eval_code(common_q, quote(y <- x * 2))
z_q <- eval_code(common_q, quote(z <- x * 3))
join_q <- join(y_q, z_q)
# get_code only has "x <- 1" occurring once
cat(get_code(join_q))
```

---

concat *Concatenate two* qenv *objects*

Description

Combine two qenv objects by simple concatenate their environments and the code.

Usage

```
concat(x, y)
```

Arguments

x	(qenv)
y	(qenv)

Details

We recommend to use the `join` method to have a stricter control in case `x` and `y` contain duplicated bindings and code. RHS argument content has priority over the LHS one.

Value

qenv object.

Examples

```
q <- qenv()
q1 <- eval_code(q, expression(iris1 <- iris, mtcars1 <- mtcars))
q2 <- q1
q1 <- eval_code(q1, "iris2 <- iris")
q2 <- eval_code(q2, "mtcars2 <- mtcars")
qq <- concat(q1, q2)
get_code(qq)
```

---

dev_suppress *Suppresses plot display in the IDE by opening a PDF graphics device*

Description

This function opens a PDF graphics device using `grDevices::pdf` to suppress the plot display in the IDE. The purpose of this function is to avoid opening graphic devices directly in the IDE.

Usage

```
dev_suppress(x)
```

Arguments

x	lazy binding which generates the plot(s)

Details

The function uses base::on.exit to ensure that the PDF graphics device is closed (using grDevices::dev.off) when the function exits, regardless of whether it exits normally or due to an error. This is necessary to clean up the graphics device properly and avoid any potential issues.

Value

No return value, called for side effects.

Examples

```
dev_suppress(plot(1:10))
```

---

eval_code *Evaluate code in* qenv

Description

Evaluate code in qenv

Usage

```
eval_code(object, code)

## S3 method for class 'qenv'
within(data, expr, ...)
```

Arguments

object	(qenv)
code	(character, language or expression) code to evaluate. It is possible to preserve original formatting of the code by providing a character or an expression being a result of parse(keep.source = TRUE).
data	(qenv)
expr	(expression) to evaluate. Must be inline code, see Using language objects...
...	named argument value will substitute a symbol in the expr matched by the name. For practical usage see Examples section below.

Details

eval_code() evaluates given code in the qenv environment and appends it to the code slot. Thus, if the qenv had been instantiated empty, contents of the environment are always a result of the stored code.

within() is a convenience method that wraps eval_code to provide a simplified way of passing expression. within accepts only inline expressions (both simple and compound) and allows to substitute expr with ... named argument values.

Value

qenv environment with code/expr evaluated or qenv.error if evaluation fails.

Using language objects with within

Passing language objects to expr is generally not intended but can be achieved with do.call. Only single expressions will work and substitution is not available. See examples.

Examples

```
# evaluate code in qenv
q <- qenv()
q <- eval_code(q, "a <- 1")
q <- eval_code(q, "b <- 2L # with comment")
q <- eval_code(q, quote(library(checkmate)))
q <- eval_code(q, expression(assert_number(a)))

# evaluate code using within
q <- qenv()
q <- within(q, {
  i <- iris
})
q <- within(q, {
  m <- mtcars
  f <- faithful
})
q
get_code(q)

# inject values into code
q <- qenv()
q <- within(q, i <- iris)
within(q, print(dim(subset(i, Species == "virginica"))))
within(q, print(dim(subset(i, Species == species)))) # fails
within(q, print(dim(subset(i, Species == species))), species = "versicolor")
species_external <- "versicolor"
within(q, print(dim(subset(i, Species == species))), species = species_external)

# pass language objects
expr <- expression(i <- iris, m <- mtcars)
within(q, expr) # fails
do.call(within, list(q, expr))

exprlist <- list(expression(i <- iris), expression(m <- mtcars))
within(q, exprlist) # fails
do.call(within, list(q, do.call(c, exprlist)))
```

get_code *Get code from* qenv

Description

Retrieves the code stored in the qenv.

Usage

```
get_code(object, deparse = TRUE, names = NULL, ...)
```

Arguments

object	(qenv)
deparse	(logical(1)) flag specifying whether to return code as character or expression.
names	(character) **[Experimental]** vector of object names to return the code for. For more details see the "Extracting dataset-specific code" section.
...	internal usage, please ignore.

Value

The code used in the qenv in the form specified by deparse.

Extracting dataset-specific code

get_code(object, names) limits the returned code to contain only those lines needed to *create* the requested objects. The code stored in the qenv is analyzed statically to determine which lines the objects of interest depend upon. The analysis works well when objects are created with standard infix assignment operators (see ?assignOps) but it can fail in some situations.

Consider the following examples:

Case 1: Usual assignments.

```
q1 <-
  within(qenv(), {
    foo <- function(x) {
      x + 1
    }
    x <- 0
    y <- foo(x)
  })
get_code(q1, names = "y")
```

x has no dependencies, so get_code(data, names = "x") will return only the second call.
y depends on x and foo, so get_code(data, names = "y") will contain all three calls.

Case 2: Some objects are created by a function's side effects.

```
q2 <-
  within(qenv(){
    foo <- function() {
      x <<- x + 1
    }
    x <- 0
    foo()
    y <- x
  })
get_code(q2, names = "y")
```

Here, `y` depends on `x` but `x` is modified by `foo` as a side effect (not by reassignment) and so `get_code(data, names = "y")` will not return the `foo()` call.
To overcome this limitation, code dependencies can be specified manually. Lines where side effects occur can be flagged by adding "`# @linksto <object name>`" at the end.
Note that `within` evaluates code passed to `expr` as is and comments are ignored. In order to include comments in code one must use the `eval_code` function instead.

```
q3 <-
  eval_code(qenv(), "
    foo <- function() {
      x <<- x + 1
    }
    x <- 0
    foo() # @linksto x
    y <- x
  ")
get_code(q3, names = "y")
```

Now the `foo()` call will be properly included in the code required to recreate `y`.

Note that two functions that create objects as side effects, `assign` and `data`, are handled automatically.

Here are known cases where manual tagging is necessary:

- non-standard assignment operators, *e.g.* `%<>%`
- objects used as conditions in `if` statements: `if (<condition>)`
- objects used to iterate over in `for` loops: `for(i in <sequence>)`
- creating and evaluating language objects, *e.g.* `eval(<call>)`

Examples

```
# retrieve code
q <- within(qenv(), {
  a <- 1
  b <- 2
})
get_code(q)
get_code(q, deparse = FALSE)
```

```
get_code(q, names = "a")

q <- qenv()
q <- eval_code(q, code = c("a <- 1", "b <- 2"))
get_code(q, names = "a")
```

---

get_env *Access environment included in* qenv

Description

The access of environment included in the qenv that contains all data objects.

Usage

```
get_env(object)
```

Arguments

object	(qenv).

Value

An environment stored in qenv with all data objects.

Examples

```
q <- qenv()
q1 <- within(q, {
  a <- 5
  b <- data.frame(x = 1:10)
})
get_env(q1)
```

---

get_messages *Get messages from* qenv *object*

Description

Retrieve all messages raised during code evaluation in a qenv.

Usage

```
get_messages(object)
```

Arguments

object	(qenv)

Value

character containing warning information or NULL if no messages.

Examples

```
data_q <- qenv()
data_q <- eval_code(data_q, "iris_data <- iris")
warning_qenv <- eval_code(
  data_q,
  bquote(p <- hist(iris_data[, .("Sepal.Length")], ff = ""))
)
cat(get_messages(warning_qenv))
```

---

get_var *Get object from* qenv

Description

[Deprecated] Instead of get_var() use native R operators/functions: x[[name]], x$name or get(): Retrieve variables from the qenv environment.

Usage

```
get_var(object, var)

## S3 method for class 'qenv.error'
x[[i]]
```

Arguments

object, x	(qenv)
var, i	(character(1)) variable name.

Value

The value of required variable (var) within qenv object.

Examples

```
q <- qenv()
q1 <- eval_code(q, code = quote(a <- 1))
q2 <- eval_code(q1, code = "b <- a")
get_var(q2, "b")
```

get_warnings *Get warnings from* qenv *object*

Description

Retrieve all warnings raised during code evaluation in a qenv.

Usage

```
get_warnings(object)
```

Arguments

object	(qenv)

Value

character containing warning information or NULL if no warnings.

Examples

```
data_q <- qenv()
data_q <- eval_code(data_q, "iris_data <- iris")
warning_qenv <- eval_code(
  data_q,
  bquote(p <- hist(iris_data[, .("Sepal.Length")], ff = ""))
)
cat(get_warnings(warning_qenv))
```

qenv *Instantiates a* qenv *environment*

Description

[Stable]

Instantiates a qenv environment.

Usage

```
qenv()
```

Details

qenv class has following characteristics:

- It inherits from the environment and methods such as $, get(), ls(), as.list(), parent.env() work out of the box.
- qenv is a locked environment, and data modification is only possible through the eval_code() and within.qenv() functions.
- It stores metadata about the code used to create the data (see get_code()).
- It supports slicing (see subset-qenv)
- It is immutable which means that each code evaluation does not modify the original qenv environment directly. See the following code:

```
q1 <- qenv()
q2 <- eval_code(q1, "a <- 1")
identical(q1, q2) # FALSE
```

Value

qenv environment.

See Also

eval_code(), get_var(), subset-qenv, get_env(),get_warnings(), join(), concat()

Examples

```
q <- qenv()
q2 <- within(q, a <- 1)
ls(q2)
q2$a
```

---

show,qenv-method *Display* qenv *object*

Description

Prints the qenv object.

Usage

```
## S4 method for signature 'qenv'
show(object)
```

Arguments

object	(qenv)

Value

`object`, invisibly.

Examples

```
q <- qenv()
q1 <- eval_code(q, expression(a <- 5, b <- data.frame(x = 1:10)))
q1
```

`subset-qenv` *Subsets* `qenv`

Description

Subsets `qenv` environment and limits the code to the necessary needed to build limited objects.

Usage

```
## S3 method for class 'qenv'
x[names, ...]
```

Arguments

`x`	(`qenv`)
`names`	(`character`) names of objects included in `qenv` to subset. Names not present in `qenv` are skipped.
`...`	internal usage, please ignore.

Examples

```
q <- qenv()
q <- eval_code(q, "a <- 1;b<-2")
q["a"]
q[c("a", "b")]
```

Index