Package 'nseval'

December 12, 2023

Type Package

Title Tools for Lazy and Non-Standard Evaluation

Version 0.5.1

Date 2023-12-11

Author Peter Meilstrup <peter.meilstrup@gmail.com>

Maintainer Peter Meilstrup <peter.meilstrup@gmail.com>

Description Functions to capture, inspect, manipulate, and create lazy values (promises), ``...'' lists, and active calls.

License GPL (>= 2.0)

Encoding UTF-8

Imports methods

Suggests testthat (>= 3.0.0), compiler (>= 3.4), roxygen2 (>= 2.2.2), knitr (>= 1.2), plyr (>= 1.8.1), lazyeval (>= 0.2.0), stringr (>= 1.2.0), rlang, covr

Collate 'arg.R' 'caller.R' 'quo.R' 'dots.R' 'getpromise.R' 'conversions.R' 'format.R' 'force.R' 'shortcut.R' 'missing.R'

RoxygenNote 7.2.3

NeedsCompilation yes

Repository CRAN

Date/Publication 2023-12-12 05:40:03 UTC

R topics documented:

`arg` *Capture lazy variables as quotations.*

Description

`arg(x)` looks in the calling environment for the binding `x`, taken literally, and returns it as a quotation. `arg(x)` is equivalent to `unwrap(quo(x))`.

`arg_` evaluates the first element normally; `arg(x, e)` is equivalent to `arg_(quote(x), e)`.

`arg_list` looks up multiple variables, and returns a dots object. `arg_list(x, y)` is equivalent to `unwrap(dots(x=x, y=y))`. If any of the requested variables are not bound, an error will be raised.

`arg_list_` is a normally evaluating version of `arg_list`; `arg_list(x, y)` is equivalent to `arg_list_(alist(x, y), environment())`.

`set_arg` and set_arg_ create bindings from quotations. They replace base function delayedAssign.

Usage

```
arg(sym, env = arg_env_(quote(sym), environment()))

arg_(sym, env = arg_env(sym, environment()))

arg_list(...)

arg_list_(syms, envs)

set_arg(dst, src)

set_arg_(dst, src)
```

Arguments

`sym`	The name to look up. For `arg` this is a literal name, not evaluated. For `arg_` this should evaluate to a symbol or character.
`env`	The environment to look in. By default, the environment from which `sym` was passed.
`...`	Bare names (not forced). Arguments may be named; these names determine the names on the output list. If arguments are not named, the names given are used.
`syms`	A character vector or list of names.
`envs`	An environment, or a list of environments, to look for the bindings in.
`dst`	A name; for `set_arg` this is quoted literally; for `set_arg_` this should be a quotation.
`src`	A quotation (or something that can be converted to a quotation, like a formula).

Value

`arg` returns a quotation object.

`args` returns a dots object.

`arg_list` returns a dots object.

Note

If you use a a literal character value, as in `arg_("x", environment())`, you MUST also give the environment parameter. The reason is that R will discard scope information about code literals in byte-compiled code; so when `arg_("x")` is called in compiled code, the default value for `env` will be found to be `emptyenv()`.

Beware of writing `arg_list(a, b, ...)` which probably doesn't do what you want. This is because R unwraps `...` before invoking `arg_list`, so this ends up double-unwrapping `...`. To capture `...` alongside named arguments you can use the syntax `arg_list(x, y, (...))` (which is equivalent to `c(arg_list(x, y), dots(...))`). You can also use `get_call()` to extract all function inputs to an active function.

See Also

dots get_dots unwrap

`arg_env` *Get information about currently bound arguments.*

Description

These are shortcut methods for querying current bindings. For example, `arg_env(x)` is equivalent to `env(arg(x))`, `is_forced(x, y)` is equivalent to `forced(arg_list(x,y))`, `dots_exprs(...)` is equivalent to `exprs(dots(...))`, and so on. The shortcut forms skip the construction of the intermediate quotation objects.

`dots_exprs(...)` quotes its arguments and returns a list of expressions. It is equivalent to `exprs(dots(...))` (and is nearly equivalent to `alist(...)`, one difference being that dots_exprs will expand `...`.)

`is_literal(x)` returns TRUE if an argument `x` could be a source literal. Specifically it tests whether `x` is bound to a singleton vector or a missing_value. This check happens without forcing `x`.

`is_missing(...)` checks whether an argument is missing, without forcing. It is similar to missing but can take multiple arguments, and can be called in more situations, such as from a nested inner function.

`is_missing_(syms, envs)` is a normally evaluating version of is_missing. `syms` should be a symbol, character vector or list of such. `envs` should be an environment, or list of environments. Vector recycling rules apply, so you can call with a vector of names and one env, or vice versa.

`is_promise` returns TRUE if the given variable is bound to a promise. Not all arguments are bound to promises; byte-compiled code often omits creating a promise for literal or missing arguments.

`is_default` determines whether an argument is bound to the function's default value for that argument. It must be called before the arguments have been forced (afterwards it will return FALSE).

Usage

```
arg_env(sym, env = arg_env_(quote(sym), environment()))

arg_env_(sym, env = arg_env_(quote(sym), environment()))

arg_expr(sym, env = arg_env_(quote(sym), environment()))

arg_expr_(sym, env = arg_env_(quote(sym), environment()))

arg_value(
  sym,
  env = arg_env_(quote(sym), environment()),
  ifnotforced = stop("Variable is not forced, so has no value")
)

arg_value_(
  sym,
  env = arg_env_(quote(sym), environment()),
  ifnotforced = stop("Variable is not forced, so has no value")
)

dots_envs(...)

dots_exprs(...)
```

```
is_forced(...)

is_forced_(syms, envs)

is_literal(...)

is_literal_(syms, envs)

is_missing(...)

is_missing_(syms, envs, unwrap = TRUE)

## S3 method for class 'quotation'
is_missing_(syms, ..., unwrap = TRUE)

is_promise(...)

is_promise_(syms, envs)

## S3 method for class 'quotation'
is_promise_(syms, ...)

is_default(...)

is_default_(syms, envs)

## S3 method for class 'quotation'
is_default_(syms, ...)
```

Arguments

`sym`	For plain `arg_env`, etc, a bare name, which is quoted. For the underscore versions `arg_env_`, something that evaluates to a name or character.
`env`	The environment to search in.
`ifnotforced`	What to return if calling arg_value on a promise that has not been forced.
`...`	Bare variable names (for `is_*`) or expressions (for `dots_*`). Not forced.
`syms`	A character vector or list of symbols.
`envs`	An environment or list of environments.
`unwrap`	Whether to recursively unwrap before testing for missingness.

Details

Throughout this package, some functions come in two forms, a "bare" version which quotes its first argument literally, and a normally-evaluating version with a trailing underscore in its name. So `is_forced(x)` chiecks whether "x" is a missing variable, while `is_forced_(x, environment())` checks whether "x" contains the *name* of another variable which is missing. The following are all equivalent:

- `arg_env(x)`
- `{y <- quo(x); arg_env_(y)}`
- `arg_env_(quote(x), environment())`
- `arg_env_(quo(x))`
- `env(arg_(quo(x)))`.

When used with quotation objects, the `is_*_` functions with trailing underscore work at one level of indirection compared to quotation methods. For example, `missing_(x)` tests whether `expr(x)` is `[missing_value()]`, whereas `is_missing_(x)` assumes `expr(x)` is a *name* and checks if that name refers to a variable that is missing. The following are equivalent:

- `is_missing(x)`
- `is_missing_(quo(x))`
- `missing_(arg(x))`

When used with a `quotation` or `dots`, is_missing(q) looks for the variable(s) specified by expr(q) in environment env(q)]‘.

Value

`arg_env` returns an environment.

`arg_expr` returns the expression bound to a named argument.

`arg_value` returns the value bound to a named argument.

`is_forced` and other `is_*` return a logical vector with optional names.

`as.dots` *Convert items into quotations or dots.*

Description

`as.dots` is a generic function for converting data into dots.

`as.dots.environment` is a synonym for `env2dots`.

Usage

```
as.dots(x)

## S3 method for class 'dots'
as.dots(x)

## S3 method for class 'quotation'
as.dots(x)

## S3 method for class 'list'
as.dots(x)
```

```
## S3 method for class 'environment'
as.dots(x)

## S3 method for class 'lazy_dots'
as.dots(x)

## Default S3 method:
as.dots(x)
```

Arguments

x	a vector or list.

Value

An object of class `...`.

See Also

env2dots rdname dots2env

`as.lazy_dots.dots` *Compatibility conversions.*

Description

Convert quotations and dot lists to representations used by some other packages.

Usage

```
as.lazy_dots.dots(x, env = "ignored")

as.lazy.quotation(x, env = "ignored")

## S3 method for class 'quosure'
as.quo(x)

## S3 method for class 'formula'
as.quo(x)

as.quosure.quo(x)

as.quosures.dots(x)
```

Arguments

x	an object to convert.
env	Ignored for quotations or dots.

Value

The converted object.

`caller` *Find the caller of a given environment.*

Description

Given an environment that is currently on the stack, `caller` determines the calling environment.

Usage

```
caller(
  env = caller(environment()),
  ifnotfound = stop("caller: environment not found on stack"),
  n = 1
)
```

Arguments

`env`	The environment whose caller to find. The default is `caller`'s caller; that is, `caller()` should return the the same value as `caller(environment())`.)
`ifnotfound`	What to return in case the caller cannot be determined. By default an error is raised.
`n`	How many levels to work up the stack.

Details

For example, in the code:

```
X <- environment()
F <- function() {
  Y <- environment()
  caller(Y)
}
F()
```

the environment called `Y` was created by calling `F()`, and that call occurs in the environment called `X`. In this case `X` is the calling environment of `Y`, so `F()` returns the same environment as `X()`.

`caller` is intended as a replacement for parent.frame, which returns the next environment up the calling stack – which is sometimes the same value, but differs in some situations, such as when lazy evaluation re-activates an environment. `parent.frame()` can return different things depending on the order in which arguments are evaluated, without warning. `caller` will by default throw an error if the caller cannot be determined.

In addition, `caller` tries to do the right thing when the environment was instantiated by means of `do.call`, eval or do rather than an ordinary function call.

Value

The environment which called env into being. If that environment cannot be determined, ifnotfound is returned.

Examples

```
E <- environment()
F <- function() {
 Y <- environment()
 caller(Y)
}
identical(F(), E) ## TRUE
```

do *Making function calls, with full control of argument scope.*

Description

The functions do and do_ construct and invoke a function call. In combination with dots and quotation objects they allow you to control the scope of the function call and each of its arguments independently.

Usage

```
do(...)

do_(...)
```

Arguments

...	A function to call and list(s) of arguments to pass. All should be quotation or dots objects, except the first argument for do which is quoted literally.

Details

For do_ all arguments should be quotation or dots objects, or convertible to such using as.quo(). They will be concatenated together by c.dots to form the call list (a dots object). For do the first argument is quoted literally, but the rest of the arguments are evaluated the same way as do_.

The head, or first element of the call list, represents the function, and it should evaluate to a function object. The rest of the call list is used as that function's arguments.

When a quotation is used as the first element, the call is evaluated from the environment given in that quotation. This means that calls to caller() (or parent.frame()) from within that function should return that environment.

do is intended to be a replacement for base function do.call. For instance these two lines are similar in effect:

```
do.call("complex", list(imaginary = 1:3))
do(complex, dots(imaginary = 1:3))
```

As are all these:

```
do.call("f", list(as.name("A")), envir = env)
do_(quo(f, env), quo(A, env)):
do_(dots_(list(as.name("f"), as.name("A")), env))
do_(dots_(alist(f, A), env))
```

Value

The return value of the call.

Note

When the environment of the call head differs from that of the arguments, do may make a temporary binding of `...` to pass arguments. This will cause some primitive functions, like (`<-`, or `for`), to fail with an error like "'...' used an in incorrect context," because these primitives do not understand how to unpack `...`. To avoid the use of `...`, ensure that all args have the same environment as the call head, or are forced.

For the specific case of calling <-, you can use `set_` to make assignments.

See Also

get_call do.call match.call set_

`dots` *Dots objects: lists of quotations.*

Description

`d <- dots(a = one, b = two)` captures each of its arguments, unevaluated, in a dots object (a named list of quotations).

`as.data.frame.dots` transforms the contents of a dots object into a data frame with one row per quotation, with columns:

- `name`: a character,
- `expr`: an expression,
- `env`: an environment object or NULL if forced,
- `value`: NULL or a value if forced.

`forced_dots(...)` forces its arguments and constructs a `dots` object with forced quotations.

`forced_dots_(values)` creates a dots object from a list of values

Usage

```
dots(...)

dots_(exprs, envs)

exprs(d)

## S3 method for class 'dots'
exprs(d)

exprs(d) <- value

## S3 replacement method for class 'dots'
exprs(d) <- value

envs(d)

## S3 method for class 'dots'
envs(d)

envs(d) <- value

## S3 method for class 'dots'
x[..., drop = FALSE]

## S3 replacement method for class 'dots'
x[...] <- value

## S3 method for class 'dots'
c(...)

## S3 method for class 'quotation'
c(...)

## S3 method for class 'dots'
as.data.frame(x, row.names = NULL, ...)

forced_dots(...)

forced_dots_(values)
```

Arguments

`...`	Any number of arguments.
`exprs`	An expression or list of expressions.
`envs`	An environment or list of environments.
`d`	A dots object.

`value`	A replacement value or list of values.
`x`	A `dots` object.
`drop`	See Extract.
`row.names`	If not given, uses `make.unique(x$name)`
`values`	A list; each element will be used as data.

Details

Objects of class "dots" mirror R's special variable `...`. Unlike `...`, a `dots` is:

- immutable (evaluating does not change it),
- first-class (you can give it any name, not just `...`),
- data (The R interpreter treates it as literal data rather than triggering argument splicing).

`d <- dots(...)` is used to capture the contents of `...` without triggering evaluation. This improves on `as.list(substitute(...()))` by capturing the environment of each argument along with their expressions. (You can also use `get_dots()`.)

Value

`dots(...)` constructs a list with class 'dots', each element of which is a quotation.

`dots_(exprs, envs)` constructs a dots object given lists of expressions and environments. (To construct a dots object from quotation objects, use `c()`.)

`exprs(d)` extracts a list of expressions from a dots object.

The mutator `exprs(d) <- value` returns a new dots object with the new expressions.

`envs(d)` extracts a list of environments from a dots object.

`envs(d)` returns a named list of environments.

`envs(d) <- value` returns an updated dots object with the environments replaced with the new value(s).

`as.data.frame.dots` returns a data frame.

Note

The columns have a class `"oneline"` for better printing.

Examples

```
named.list <- function(...) {
 # Collect only named arguments, ignoring unnamed arguments.
 d <- dots(...)
 do(list, d[names(d) != ""])
}

named.list(a=1, b=2*2, stop("this is not evaluated"))
```

dots2env	*Make or update an environment with bindings from a dots list.*

Description

All named entries in the dots object will be bound to variables. Unnamed entries will be appended to any existing value of `...` in the order in which they appear.

Usage

```
dots2env(
  x,
  env = new.env(hash = hash, parent = parent, size = size),
  names = NULL,
  use_dots = TRUE,
  append = TRUE,
  hash = (length(dots) > 100),
  size = max(29L, length(dots)),
  parent = emptyenv()
)

## S3 method for class 'dots'
as.environment(x)
```

Arguments

`x`	A dots object with names.
`env`	Specify an environment object to populate and return. By default a new environment is created.
`names`	Which variables to populate in the environment. If NULL is given, will use all names present in the dotlist. If a name is given that does not match any names from the dots object, an error is raised.
`use_dots`	Whether to bind unnamed or unmatched items to `...`. If FALSE, these items are discarded. If TRUE, they bound to `...` in the environment. If items have duplicate names, the earlier ones are used and the rest placed in "...".
`append`	if `TRUE`, unmatched or unnamed items will be appended to any existing value of '...'. If `FALSE`, the existing binding of `...` will be cleared. (Neither happens if `use_dots` is FALSE.)
`hash`	if env is NULL, this argument is passed to `new.env`.
`size`	if env is NULL, this argument is paseed to `new.env`.
`parent`	if env is NULL, this argument is paseed to `new.env`.

Value

An environment object.

See Also

env2dots

`env2dots` *Copy bindings from an environment into a dots object, or vice versa.*

Description

`env2dots` copies all bindings in the environment (but not those from its parents) into a new dots object. Bindings that are promises will be captured without forcing. Bindings that are not promises will be rendered as forced quotations. The output will not be in any guaranteed order.

Usage

```
env2dots(
  env = caller(environment()),
  names = ls(envir = env, all.names = TRUE),
  include_missing = TRUE,
  expand_dots = TRUE
)
```

Arguments

`env`	An environment.
`names`	Which names to extract from the environment. By default extracts all bindings present in `env`, but not in its enclosing environments.
`include_missing`	Whether to include missing bindings.
`expand_dots`	Whether to include the contents of `...`.

Value

A dots object.

`forced` *Forcing and forcedness of arguments and quotations.*

Description

There are two kinds of quotations: forced and unforced. Unforced quotations have an expression and an environment; forced quotations have an expression and a value.

`forced(q)` tests whether a quotation is forced.

`forced(d)` on a dots object tests whether each element is forced, and returns a logical vector.

`force_(x)` converts an unforced quotation or dots object into a forced one, by evaluating it.

`value(x)` or `values(...)` returns the value of a quotation or dots, forcing it if necessary.

Usage

```
forced(x)

## S3 method for class 'quotation'
forced(x, ...)

## S3 method for class 'dots'
forced(x)

## Default S3 method:
forced(x)

force_(x, ...)

## S3 method for class 'quotation'
force_(x, eval = base::eval, ...)

## S3 method for class 'dots'
force_(x, ...)

value(x, ...)

## S3 method for class 'quotation'
value(x, ...)

## S3 method for class 'dots'
value(x, ...)

values(x)

## S3 method for class 'dots'
values(x)
```

Arguments

`x`	A quotation or dots object.
`...`	Options used by methods
`eval`	Which evaluation function to use.

Value

`forced(x)` returns a logical.

`value(x)` returns the result of forcing the quotation.

`values` returns a list.

See Also

is_forced forced_quo

force

`format.dots` *Formatting methods for dots and quotations.*

Description

`format.dots` constructs a string representation of a dots object.

`format.quotation` constructs a string representation of a quotation object.

`format.oneline` formats a vector or list so that each item is displayed on one line. It is similar to format.AsIs but tries harder witlanguage objects. The "oneline" class is used by as.data.frame.dots.

Usage

```
## S3 method for class 'dots'
format(
  x,
  compact = FALSE,
  show.environments = !compact,
  show.expressions = !compact,
  width = NULL,
  ...
)

## S3 method for class 'quotation'
format(
  x,
  compact = FALSE,
  show.environments = !compact,
  show.expressions = !compact,
  width = NULL,
  ...
)

## S3 method for class 'oneline'
format(x, max.width = 50, width = max.width, ...)

## S3 method for class 'dots'
print(x, ...)

## S3 method for class 'quotation'
print(x, ...)
```

Arguments

x	An object.

`compact`	Implies `show.environments=FALSE` and `show.expressions=FALSE`.
`show.environments`	Whether to show environments for unforced quotations.
`show.expressions`	Whether to show expressions for forced quotations.
`width`	See base::format.
`...`	Further parameters passed along to base::format.
`max.width`	See base::format.

`function_` *Explicitly create closures.*

Description

`function_` is a normally-evaluating version of `function`, which creates closures. A closure object has three components: the argument list, the body expression, and the enclosing environment.

`arglist()` is a helper that produces a named list of missing_values given a character vector of names.

Usage

```
function_(args, body, env = arg_env(args, environment()))

arglist(names, fill = missing_value())
```

Arguments

`args`	The argument list of the new function. NULL is accepted to make a function with no arguments. Arguments are specified as a named list; the list names become the argument names, and the list values become the default expressions. A value of `missing_value()` indicates no default. alist and arglist are useful for making argument lists.
`body`	An expression for the body of the function.
`env`	The enclosing environment of the new function.
`names`	A character vector.
`fill`	The expression (default missing)

Value

A closure.

See Also

environment formals body

Examples

```
f1 <- function(x, y = x) { x + y }
f2 <- function_(alist(x = , y = x),
                quote( { x + y } ),
                environment())
identical(f1, f2) # TRUE

# `fn` makes a compact way to write functions;
# `fn(x+y)` is equivalent to `function(x, y) x+y`
fn <- function(exp) {
  exp_ <- arg(exp)
  nn <- arglist(all.names(expr(exp_), functions=FALSE))
  function_(nn, expr(exp_), env(exp_))
}

fn(x^2)
fn(x+y)
```

---

get_call *Get information about currently executing calls.*

Description

get_call(env), given an environment associated with a currently executing call, returns the function call and its arguments, as a dots object. To replicate a call, the dots object returned can be passed to do.

get_function(env) finds the function object associated with a currently executing call.

Usage

```
get_call(
  env = caller(environment()),
  ifnotfound = stop("get_call: environment not found on stack")
)

get_function(
  env = caller(environment()),
  ifnotfound = stop("get_function: environment not found on stack")
)
```

Arguments

env	An environment belonging to a currently executing function call. By default, the caller of get_call itself (so get_call() is equivalent to get_call(environment()).)
ifnotfound	What to return if the call is not found. By default an error is thrown.

Details

get_call is meant to replace match.call and sys.call; its advantage is that it captures the environments bound to arguments in addition to their written form.

get_function is similar to sys.function, but is keyed by environment rather than number.

Value

get_call returns a dots object, the first element of which represents the function name and caller environment.

get_function returns a closure.

See Also

do dots caller

Examples

```
# We might think of re-writing the start of [lm] like so:
LM <- function(formula, data, subset, weights, na.action, method = "qr",
               model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
               contrasts = NULL, offset, ...) {
  cl <- get_call()
  mf <- do(model.frame,
           arg_list(formula, data, subset, weights, na.action, offset))

  z <- get.call()

  class(z) <- c("LM", class(z))
  z$call <- cl
  z
}

# and `update` like so:
update.LM <- function(object, formula., ...) {
  call <- object$call
  extras <- dots(...)
  call$formula <- forced_quo(update.formula(formula(object), formula.))
  do(call)
}
```

---

get_dots *Set or get the contents of ....*

Description

get_dots() unpacks ... from a given environment and returns a dots object.

set_dots takes a dots list and uses it to create a binding for ... in a given environment.

Usage

```
get_dots(env = caller(environment()), inherits = FALSE)

set_dots(env, d, append = FALSE)
```

Arguments

`env`	The environment to look in.
`inherits`	Whether to pull `...` from enclosing environments.
`d`	a `[dots]` object.
`append`	if TRUE, the values should be appended to the existing binding. If false, existing binding for "..." will be replaced.

Details

`get_dots()` is equivalent to `dots(...)` or ``arg_list(`...`)``.

Value

`get_dots` returns a dots list. If `...` is not bound or is missing, it returns an empty dots list.

`set_dots` returns the updated environment, invisibly.

See Also

env2dots set_arg dots2env

`locate` *Find the environment which defines a name.*

Description

`locate` starts at a given environment, and searches enclosing environments for a name. It returns the first enclosing environment which defines `sym`.

`locate_` is the normally evaluating method; `locate(x)` is equivalent to `locate_(quo(x))` or `locate_(quote(x), environment())`.

When `sym` is a quotation or dots, any `env` argument is ignored.

Usage

```
locate(sym, env = arg_env_(quote(sym), environment()), mode = "any", ...)

locate_(sym, env = arg_env_(quote(sym), environment()), mode = "any", ...)

locate_.list(sym, env = arg_env_(quote(sym), environment()), mode = "any", ...)
locate_.quotation(sym, env = "ignored", mode = "any", ...)

locate_.character(
  sym,
  env = arg_env_(quote(sym), environment()),
  mode = "any",
  ...
)

`locate_.(`(sym, env = arg_env_(quote(sym), environment()), mode = "any", ...)

locate_.dots(sym, env = "ignored", mode = "any", ...)

locate_.name(
  sym,
  env = arg_env_(quote(sym), environment()),
  mode = "any",
  ifnotfound = stop("Binding ", deparse(sym), " not found")
)
```

Arguments

`sym`	A name. For `locate` the argument is used literally. For `locate_` it should be a name or list of names.
`env`	Which environment to begin searching from.
`mode`	Either `"any"` or `"function"`. `"any"` finds the lowest enclosing environment which gives any definiton for `sym`. `"function"` searches for an environment which defines `sym` as a function. This may force lazy arguments in the process, in the same way as get.
`...`	Further arguments passed to methods.
`ifnotfound`	What is returned if the symbol is not found. By default an exception is raised.

Value

An environment object which defines `sym`, if one is found.

If `sym` is a list (of names) or a dots object, `locate_(sym)` returns a list.

Note

To locate where `...` is bound, you can wrap it in parens, as `locate( (...) )`.

If you use a literal character argument, as in `locate("x", environment())`, you must also provide the environment argument explicitly; `locate("x")` won't work in compiled functions. However using a literal name like `locate(x)` will work OK. See note under arg.

Examples

```
# Here is how to implement R's `<<-` operator, using `locate_`:
`%<<-%` <- function(lval, rval) {
  lval_ <- arg(lval)
  name <- expr(lval_)
  target.env <- locate_(name, parent.env(env(lval_)))
  assign(as.character(name), rval, envir=target.env)
}

x <- "not this one"
local({
  x <- "this one"
  local({
    x <- "not this one either"
    x %<<-% "this works like builtin <<-"
  })
  print(x)
})
print(x)
```

`missing_value` *R's missing value.*

Description

`missing_value()` returns R's missing object; what R uses to represent a missing argument. It is distinct from either NULL or NA.

Usage

```
missing_value(n)

missing_(x, unwrap = TRUE)

## Default S3 method:
missing_(x, unwrap = TRUE)

## S3 method for class 'dots'
missing_(x, unwrap = TRUE)

## S3 method for class 'quotation'
missing_(x, unwrap = TRUE)

list_missing(...)
```

Arguments

`n`	Optional; a number. If provided, will return a list of missing values with this many elements.
`x`	a value, dots, or list.

`unwrap`	Whether to descend recursively through unevaluated promises using `unwrap(x, TRUE)`
`...`	Arguments evaluated normally. except those which are missing.

Details

The missing value occurs naturally in a quoted R expression that has an empty argument:

```
exp <- quote( x[1, ] )
identical(exp[[4]], missing_value()) #TRUE
is_missing(exp[[4]]) #also TRUE
```

So we can use `missing_value()` to help construct expressions:

```
substitute(f[x, y], list(x = 1, y=missing_value()))
```

When such an expression is evaluated and starts a function call, the missing value winds up in the promise expression.

```
f <- function(x) arg_expr(x)
identical(f(), missing_value()) # TRUE
```

During "normal evaluation", finding a missing value in a variable raises an error.

```
m <- missing_value()
list(m) # raises error
```

This means that it's sometimes tricky to work with missings:

```
exp <- quote( x[1, ] )
cols <- x[[4]]
x <- list(missing_value(), 2, 3)     # this is ok, but...
a <- missing_value(); b <- 2; c <- 3 # this stores missing in "cols",
x <- list(a, b, c)                   # throws an error: "a" missing
```

Generally, keep your missing values wrapped up in lists or quotations, instead of assigning them to variables directly.

Value

`missing_value` returns the symbol with empty name, or a list of such.

`missing_` returns a logical vector.

`list_missing` returns a list.

See Also

missing is_missing

missing is_missing

Examples

```
# These expressions are equivalent:
function(x, y=1) {x+y}
function_(list(x=missing_value, y=1),
          quote( {x+y} ))

# These expressions are also equivalent:
quote(df[,1])
substitute(df[row,col],
           list(row = missing_value(), col = 1))
# How to do the trick of `[` where it can tell which arguments are
# missing:
`[.myclass` <- function(x, ...) {
   indices <- list_missing(...)
   kept.axes <- which(missing_(indices))
   cat(paste0("Keeping axes ", kept_axes, "\n"))
   #...
}
ar <- structure(array(1:24, c(2, 3, 4)))
ar[, 3, ]
```

---

quo *Quotation objects.*

Description

quo(expr, env) captures expr without evaluating, and returns a qutation object. A quotation has two parts: an expression expr(q) with an environment env(q).

quo_(expr, env) is the normally evaluating version. It constructs a quotation given an expression and environment.

as.quo(x) converts an object into a quotation. Closures, formulas, and single-element dots can be converted this way.

forced_quo(x) captures the expression in its argument, then forces it, returning a quotation with the expression and value.

forced_quo_(val) makes a forced quotation given a value. Specifically it constructs a quotation with the same object in both the expr and value slots, except if is a language object in which case the expr slot is wrapped in quote().

Usage

```
quo(expr, env = arg_env_(quote(expr), environment()), force = FALSE)

quo_(expr, env, force = FALSE)

env(q)

env(q) <- value
```

```
expr(q)

## S3 method for class 'quotation'
expr(q)

expr(q) <- value

is.quotation(x)

is.quo(x)

as.quo(x)

forced_quo(x)

forced_quo_(val)
```

Arguments

`expr`	An expression. For `quo` this is taken literally and not evaluated. For `quo_` this is evaluated normally.
`env`	An environment.
`force`	Whether to evaluate the expression and create a forced quotation.
`q`	A quotation object.
`value`	An updated value.
`x`	Any object.
`val`	A value.

Details

(Like in writing, an 'expression' may simply be a set of words, but a 'quotation' comes bundled with a citation, to reference a context in which it was said.)

A quo is parallel to a 'promise' which is the data structure R uses to hold lazily evaluated arguments. A quo is different from a promise because it is an immutable data object.

As a data object, a quo does not automatically evaluate like a promise, but can be evaluated explicitly with the methods value or force_. A quo is immutable, so it does not mutate into a "forced" state if you choose to evaluate it; instead `force_(q)` returns a new object in the forced state.

A function can capture its arguments as quotations using `arg`.

A dots object is a list of quotations.

Value

`quo_` and `quo` return an object of class "quotation".

`as.quo` returns a quotation.

set_ *Assign values to variables.*

Description

`set_` is a normally-evaluating version of `<-`. `set_enclos_` is a normally evaluating version of `<<-`.

Usage

```
set_(dest, val)

set_enclos_(dest, val)
```

Arguments

`dest`	A quotation specifying the destination environment and name. This can also be an indexing, expression, and `set_` will perform subassignment.
`val`	The value to assign.

Details

`set_` differs from `[assign]` in that `set_` will process subassignments.

These helpers are here because it is tricky to use `do_` with `<-` (see Note under do_).

Value

`set_` returns `val`, invisibly.

Examples

```
set_(quo(x), 12) #equivalent to `x <- 12`
set_(quo(x[3]), 12) #equivalent to `x[3] <- 12`
e <- new.env()
set_(quo(x[3], e), 12) #assigns in environment `e`
set_enclos_(quo(x[3], e), 12) #assigns in a parent of environment `e`
```

unwrap *Unwrap variable references.*

Description

Given an unforced quotation whose expression is a bare variable name, `unwrap` follows the variable reference, and returns a quotation. When the argument is forced or has a nontrivial expression `unwrap` has no effect.

Usage

```
unwrap(x, recursive = FALSE)

## S3 method for class 'dots'
unwrap(x, recursive = FALSE)
```

Arguments

`x`	a quotation to unwrap.
`recursive`	Default `FALSE` unwraps exactly once. If `TRUE`, unwrap as far as possible (until a forced promise or nontrivial expression is found.)

Details

There are two good use cases for `unwrap(x, recursive=TRUE)`. One is to derive plot labels (the most inoccuous use of metaprogramming). Another is to check for missingness (this is what R's missing and does as well).

Using `unwrap(x, recursive=TRUE)` in other situations can get you into confusing situations – effectively you are changing the behavior of a parent function that may be an unknown number of levels up the stack, possibly turning a standard-evaluating function into nonstandard-evaluating function. So recursive unerapping is not the default behavior.

Value

The quotation method returns a quotation.

The dots method returns a dots object with each quotation unwrapped.

Examples

```
# different levels of unwrapping:
f <- function(x) { g(x) }
g <- function(y) { h(y) }
h <- function(z) {
  print(arg(z))
  print(unwrap(quo(z)))
  print(unwrap(unwrap(quo(z))))
  print(unwrap(quo(z), recursive=TRUE))
}

w <- 5
f(w)
```

Index