Package ‘lazyeval’

October 29, 2017

Version 0.2.1
Title Lazy (Non-Standard) Evaluation
Description An alternative approach to non-standard evaluation using formulas. Provides a full implementation of LISP style 'quasiquotation', making it easier to generate code with other code.
License GPL-3
LazyData true
Depends R (>= 3.1.0)
Suggests knitr, rmarkdown (>= 0.2.65), testthat, covr
VignetteBuilder knitr
RoxygenNote 6.0.1
NeedsCompilation yes
Author Hadley Wickham [aut, cre], RStudio [cph]
Maintainer Hadley Wickham <hadley@rstudio.com>
Repository CRAN
Date/Publication 2017-10-29 22:59:36 UTC

R topics documented:

  as.lazy ................................. 2
  ast_ ..................................... 3
  as_name .................................. 3
  call_modify ............................... 4
  call_new .................................. 5
  expr_label ............................... 5
  function_new ............................. 6
  f_capture ................................ 7
  f_eval_rhs ................................ 8
  f_interp .................................. 9
  f_list .................................... 10
as.lazy

Convert an object to a lazy expression or lazy dots.

Description

Convert an object to a lazy expression or lazy dots.

Usage

```r
as.lazy(x, env = baseenv())

as.lazy_dots(x, env)
```

Arguments

- **x**: An R object. Current methods for `as.lazy()` convert formulas, character vectors, calls and names. Methods for `as.lazy_dots()` convert lists and character vectors (by calling `lapply()` with `as.lazy()`).
- **env**: Environment to use for objects that don’t already have associated environment.

Examples

```r
as.lazy(~ x + 1)
as.lazy(quote(x + 1), globalenv())
as.lazy("x + 1", globalenv())

as.lazy_dots(list(~x, y = ~z + 1))
as.lazy_dots(c("a", "b", "c"), globalenv())
as.lazy_dots(~x)
as.lazy_dots(quote(x), globalenv())
as.lazy_dots(quote(f()), globalenv())
as.lazy_dots(lazy(x))
```
**ast_**

Display a call (or expression) as a tree.

**Description**

ast_ takes a quoted expression; ast does the quoting for you.

**Usage**

```r
ast_(x, width = getOption("width"))
```

```
ast(x)
```

**Arguments**

- `x` Quoted call, list of calls, or expression to display.
- `width` Display width, defaults to current width as reported by `getOption("width")`.

**Examples**

```r
ast(f(x, 1, g(), h(i())))
ast(if (TRUE) 3 else 4)
ast(function(a = 1, b = 2) (a + b + 10))
ast(f(x)(y)(z))

ast_(quote(f(x, 1, g(), h(i()))))
ast_(quote(if (TRUE) 3 else 4))
ast_(expression(1, 2, 3))
```

**as_name**

Coerce an object to a name or call.

**Description**

These are a S3 generics with built-in methods for names, calls, formulas, and strings. The distinction between a name and a call is particularly important when coercing from a string. Coercing to a call will parse the string, coercing to a name will create a (potentially) non-syntactic name.

**Usage**

```r
as_name(x)
```

```
as_call(x)
```

**Arguments**

- `x` An object to coerce
call_modify

Modify the arguments of a call.

Description

Modify the arguments of a call.

Usage

```r
call_modify(call, new_args, env = parent.frame())
call_standardise(call, env = parent.frame())
```

Arguments

- `call`: A call to modify. It is first standardised with `call_standardise`.
- `new_args`: A named list of expressions (constants, names or calls) used to modify the call. Use `NULL` to remove arguments.
- `env`: Environment in which to look up call value.

Examples

```r
call <- quote(mean(x, na.rm = TRUE))
call_standardise(call)

# Modify an existing argument
call_modify(call, list(na.rm = FALSE))
call_modify(call, list(x = quote(y))))

# Remove an argument
call_modify(call, list(na.rm = NULL))

# Add a new argument
call_modify(call, list(trim = 0.1))

# Add an explicit missing argument
call_modify(call, list(na.rm = quote(expr = )))
```
**call_new**  Create a call by "hand"

**Description**

Create a call by "hand"

**Usage**

```r
call_new(f, ..., .args = list())
```

**Arguments**

- `f`  
  Function to call. For `make_call`, either a string, a symbol or a quoted call. For `do_call`, a bare function name or call.
- `...`, `.args`  
  Arguments to the call either in or out of a list

**Examples**

```r
# f can either be a string, a symbol or a call
call_new("f", a = 1)
call_new(quote(f), a = 1)
call_new(quote(f()), a = 1)

#' Can supply arguments individually or in a list
call_new(quote(f), a = 1, b = 2)
call_new(quote(f), .args = list(a = 1, b = 2))
```

**expr_label**  Find the expression associated with an argument

**Description**

`expr_find()` finds the full expression; `expr_text()` turns the expression into a single string; `expr_label()` formats it nicely for use in messages. `expr_env()` finds the environment associated with the expression.

**Usage**

```r
expr_label(x)
expr_text(x, width = 60L, nlines = Inf)
expr_find(x)
expr_env(x, default_env)
```
function_new

Create a function by "hand"

Description

This constructs a new function given it’s three components: list of arguments, body code and parent environment.

Usage

function_new(args, body, env = parent.frame())
**f_capture**

**Arguments**

- **args**: A named list of default arguments. Note that if you want arguments that don’t have defaults, you’ll need to use the special function `alist`, e.g. `alist(a = , b = 1)`.
- **body**: A language object representing the code inside the function. Usually this will be most easily generated with `quote`.
- **env**: The parent environment of the function, defaults to the calling environment of `make_function`.

**Examples**

```r
f <- function(x) x + 3
g <- function_new(alist(x = ), quote(x + 3))

# The components of the functions are identical
identical(formals(f), formals(g))
identical(body(f), body(g))
identical(environment(f), environment(g))

# But the functions are not identical because f has src code reference
identical(f, g)
attr(f, "srcref") <- NULL
# Now they are:
stopifnot(identical(f, g))
```

**Description**

This should be used sparingly if you want to implement true non-standard evaluation with 100% magic. I recommend avoiding this unless you have strong reasons otherwise since requiring arguments to be formulas only adds one extra character to the inputs, and otherwise makes life much much simpler.

**Usage**

```r
f_capture(x)

dots_capture(..., .ignore_empty = TRUE)
```

**Arguments**

- **x, ...**: An unevaluated promises
- **.ignore_empty**: If TRUE, empty arguments will be silently dropped.
Value

f_capture returns a formula; dots_capture returns a list of formulas.

Examples

f_capture(a + b)
dots_capture(a + b, c + d, e + f)

# These functions will follow a chain of promises back to the
# original definition
f <- function(x) g(x)
g <- function(y) h(y)
h <- function(z) f_capture(z)
f(a + b + c)

---

f_eval_rhs Evaluate a formula

Description

f_eval_rhs evaluates the RHS of a formula and f_eval_lhs evaluates the LHS. f_eval is a shortcut for f_eval_rhs since that is what you most commonly need.

Usage

f_eval_rhs(f, data = NULL)
f_eval_lhs(f, data = NULL)
f_eval(f, data = NULL)
find_data(x)

Arguments

f  A formula. Any expressions wrapped in \texttt{uq()} will will be "unquoted", i.e. they will be evaluated, and the results inserted back into the formula. See \texttt{f_interp} for more details.

data  A list (or data frame). find_data is a generic used to find the data associated with a given object. If you want to make \texttt{f_eval} work for your own objects, you can define a method for this generic.

x  An object for which you want to find associated data.

Details

If data is specified, variables will be looked for first in this object, and if not found in the environment of the formula.
**Pronouns**

When used with data, \( \texttt{f_eval} \) provides two pronouns to make it possible to be explicit about where you want values to come from: \( \texttt{.env} \) and \( \texttt{.data} \). These are thin wrappers around \( \texttt{.data} \) and \( \texttt{.env} \) that throw errors if you try to access non-existent values.

**Examples**

\[
\texttt{f_eval(\sim 1 + 2 + 3)}
\]

# formulas automatically capture their enclosing environment

\[
\text{foo <- function(x) { y <- 10 \sim x + y}}
\]

\[
f <- \text{foo(1)}
f \text{f_eval(f)}
\]

# If you supply data, \texttt{f_eval} will look their first:

\[
f \text{eval(\sim \texttt{cyl}, mtcars)}
\]

# To avoid ambiguity, you can use \( \texttt{.env} \) and \( \texttt{.data} \) pronouns to be explicit:

\[
\text{cyl <- 10}
\]

\[
f \text{eval(\sim \texttt{.data$ cyl}, mtcars)}
f \text{eval(\sim \texttt{.env$ cyl}, mtcars)}
\]

# Imagine you are computing the mean of a variable:

\[
f \text{eval(\sim \texttt{mean(cyl)}, mtcars)}
\]

# How can you change the variable that's being computed?

# The easiest way is "unquote" with \( \texttt{uq()} \)

# See \texttt{?f_interp} for more details

\[
\text{var <- \sim \texttt{cyl}}
\text{f \text{eval(\sim \texttt{mean(uq(var))}, mtcars)}}
\]

---

**f_interp**  
Interpolate a formula

**Description**

Interpolation replaces sub-expressions of the form \( \texttt{uq(x)} \) with the evaluated value of \( x \), and inlines sub-expressions of the form \( \texttt{uqs(x)} \).

**Usage**

\[
f \text{interp(f, data = \texttt{NULL})}
\]

\[
\text{uq(x, data = \texttt{NULL})}
\]
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>A one-sided formula.</td>
</tr>
<tr>
<td>data</td>
<td>When called from inside f_eval, this is used to pass on the data so that nested formulas are evaluated in the correct environment.</td>
</tr>
<tr>
<td>x</td>
<td>For uq and uqf, a formula. For uqs, a a vector.</td>
</tr>
</tbody>
</table>

Theory

Formally, \( f_{\text{ interp }} \) is a quasiquote function, \( uq() \) is the unquote operator, and \( uqs() \) is the unquote splice operator. These terms have a rich history in LISP, and live on in modern languages like [http://docs.julialang.org/en/release-0.1/manual/metaprogramming/](http://docs.julialang.org/en/release-0.1/manual/metaprogramming/) and [https://docs.racket-lang.org/reference/quasiquote.html](https://docs.racket-lang.org/reference/quasiquote.html).

Examples

```
f_interp(x ~ 1 + uq(1 + 2 + 3) + 10)
```

# Use uqs() if you want to add multiple arguments to a function
# It must evaluate to a list
```
args <- list(1:10, na.rm = TRUE)
f_interp(~ mean( uqs(args) ))
```

# You can combine the two
```
var <- quote(xyz)
extra_args <- list(trim = 0.9)
f_interp(~ mean( uq(var) , uqs(extra_args) ))
```

```
foo <- function(n) {
  ~ 1 + uq(n)
}
f <- foo(10)
f
f_interp(f)
```

---

\( f_{\text{ list }} \)

Build a named list from the LHS of formulas

Description

\( f_{\text{ list }} \) makes a new list; as \( f_{\text{ list }} \) takes an existing list. Both take the LHS of any two-sided formulas and evaluate it, replacing the current name with the result.
**f_new**

*Usage*

\[
\text{f\_list}(...) \\
\text{as\_f\_list}(x)
\]

**Arguments**

- \ldots: Named arguments.
- \textit{x}: An existing list

**Value**

A named list.

**Examples**

\[
\text{f\_list}("y\sim x") \\
\text{f\_list}(a = "y\sim a, ~ b, c = ~c")
\]

---

**f\_new**  

Create a formula object by "hand".

*Description*

Create a formula object by "hand".

*Usage*

\[
\text{f\_new}(\text{rhs, lhs = NULL, env = parent.frame()})
\]

**Arguments**

- \textit{lhs, rhs}: A call, name, or atomic vector.
- \textit{env}: An environment

**Value**

A formula object

**Examples**

\[
\text{f\_new}(\text{quote(a)}) \\
\text{f\_new}(\text{quote(a), quote(b)})
\]
Description

\( f_{\text{rhs}} \) extracts the righthand side, \( f_{\text{lhs}} \) extracts the lefthand side, and \( f_{\text{env}} \) extracts the environment. All functions throw an error if \( f \) is not a formula.

Usage

\[
\begin{align*}
f_{\text{rhs}}(f) \\
f_{\text{rhs}}(x) &\leftarrow \text{value} \\
f_{\text{lhs}}(f) \\
f_{\text{lhs}}(x) &\leftarrow \text{value} \\
f_{\text{env}}(f) \\
f_{\text{env}}(x) &\leftarrow \text{value}
\end{align*}
\]

Arguments

- \( f \), \( x \) A formula
- value The value to replace with.

Value

\( f_{\text{rhs}} \) and \( f_{\text{lhs}} \) return language objects (i.e. atomic vectors of length 1, a name, or a call). \( f_{\text{env}} \) returns an environment.

Examples

\[
\begin{align*}
f_{\text{rhs}}(- 1 + 2 + 3) \\
f_{\text{rhs}}(- x) \\
f_{\text{rhs}}(- "A") \\
f_{\text{rhs}}(1 - 2) \\
f_{\text{lhs}}(- y) \\
f_{\text{lhs}}(x - y) \\
f_{\text{env}}(- x)
\end{align*}
\]
**f_text**

*Turn RHS of formula into a string/label.*

**Description**

Equivalent of `expr_text()` and `expr_label()` for formulas.

**Usage**

```
\f_text(x, width = 60L, nlines = Inf)
```

```
\f_label(x)
```

**Arguments**

- **x** A formula.
- **width** Width of each line
- **nlines** Maximum number of lines to extract.

**Examples**

```r
f <- ~ a + b + bc
f_text(f)
f_label(f)
```

```r
# Names a quoted with ``
f_label(~ x)
```

```r
# Strings are encoded
f_label(~ "a\nb")
```

```r
# Long expressions are collapsed
f_label(~ foo({
  1 + 2
  print(x)
}))
```

---

**f_unwrap**

*Unwrap a formula*

**Description**

This interpolates values in the formula that are defined in its environment, replacing the environment with its parent.

**Usage**

```
\f_unwrap(f)
```
Arguments

f  A formula to unwrap.

Examples

\begin{verbatim}
n <- 100
f <- -x + n
f_unwrap(f)
\end{verbatim}

interp  \textit{Interpolate values into an expression.}

Description

This is useful if you want to build an expression up from a mixture of constants and variables.

Usage

\begin{verbatim}
interp(`_obj`, ..., .values)
\end{verbatim}

Arguments

_obj  An object to modify: can be a call, name, formula, lazy, or a string.

..., .values  Either individual name-value pairs, or a list (or environment) of values.

Examples

\begin{verbatim}
# Interp works with formulas, lazy objects, quoted calls and strings
interp(- x + y, x = 10)
interp(lazy(x + y), x = 10)
interp(quote(x + y), x = 10)
interp("x + y", x = 10)

# Use as.name if you have a character string that gives a
# variable name
interp(~ mean(var), var = as.name("mpg"))
# or supply the quoted name directly
interp(~ mean(var), var = quote(mpg))

# Or a function!
interp(~ f(a, b), f = as.name("+"))
# Remember every action in R is a function call:
# http://adv-r.had.co.nz/Functions.html#all-calls

# If you've built up a list of values through some other
# mechanism, use .values
interp(~ x + y, .values = list(x = 10))

# You can also interpolate variables defined in the current
\end{verbatim}
is_formula

# environment, but this is a little risky.
y <- 10
interp(~ x + y, .values = environment())

is_formula Is object a formula?

Description
Is object a formula?

Usage
is_formula(x)

Arguments
x Object to test

Examples
is_formula(~ 10)
is_formula(10)

is_lang Is an object a language object?

Description
These helpers are consistent wrappers around their base R equivalents. A language object is either
an atomic vector (typically a scalar), a name (aka a symbol), a call, or a pairlist (used for function
arguments).

Usage
is_lang(x)

is_name(x)

is_call(x)

is_pairlist(x)

is_atomic(x)
Arguments

x  An object to test.

See Also

as_name() and as_call() for coercion functions.

Examples

q1 <- quote(1)
is_lang(q1)
is_atomic(q1)

q2 <- quote(x)
is_lang(q2)
is_name(q2)

q3 <- quote(x + 1)
is_lang(q3)
is_call(q3)

Description

lazy() uses non-standard evaluation to turn promises into lazy objects; lazy_H() does standard evaluation and is suitable for programming.

Usage

lazy_(expr, env)

lazy(expr, env = parent.frame(), .follow_symbols = TRUE)

Arguments

expr  Expression to capture. For lazy_, must be a name or a call.
env  Environment in which to evaluate expr.
.follow_symbols

If TRUE, the default, follows promises across function calls. See vignette("chained-promises") for details.

Details

Use lazy() like you’d use substitute() to capture an unevaluated promise. Compared to substitute(), it also captures the environment associated with the promise, so that you can correctly replay it in the future.
Examples

```r
lazy_(quote(a + x), globalenv())

# Lazy is designed to be used inside a function - you should
# give it the name of a function argument (a promise)
f <- function(x = b - a) {
  lazy(x)
}
f()
f(a + b / c)

# Lazy also works when called from the global environment. This makes
# easy to play with interactively.
lazy(a + b / c)

# By default, lazy will climb all the way back to the initial promise
# This is handy if you have if you have nested functions:
g <- function(y) f(y)
h <- function(z) g(z)
f(a + b)
g(a + b)
h(a + b)

# To avoid this behaviour, set .follow_symbols = FALSE
# See vignette("chained-promises") for details
```

Description

Capture ... (dots) for later lazy evaluation.

Usage

```r
lazy_dots(..., .follow_symbols = FALSE, .ignore_empty = FALSE)
```

Arguments

- `...`: Dots from another function
- `.follow_symbols`: If TRUE, the default, follows promises across function calls. See vignette("chained-promises") for details.
- `.ignore_empty`: If TRUE, empty arguments will be ignored.

Value

A named list of lazy expressions.
Examples

```r
lazy_dots(x = 1)
lazy_dots(a, b, c * 4)

f <- function(x = a + b, ...) {
    lazy_dots(x = x, y = a + b, ...)
}
f(z = a + b)
f(z = a + b, .follow_symbols = TRUE)

# .follow_symbols is off by default because it causes problems
# with lazy loaded objects
lazy_dots(letters)
lazy_dots(letters, .follow_symbols = TRUE)

# You can also modify a dots like a list. Anything on the RHS will
# be coerced to a lazy.
l <- lazy_dots(x = 1)
l$y <- quote(f)
l[c("y", "x")]
l["z"] <- list(g)

c(lazy_dots(x = 1), lazy_dots(f))
```

---

**lazy_eval**

Evaluate a lazy expression.

**Description**

Evaluate a lazy expression.

**Usage**

```r
lazy_eval(x, data = NULL)
```

**Arguments**

- **x**
  - A lazy object or a formula.
- **data**
  - Option, a data frame or list in which to preferentially look for variables before using the environment associated with the lazy object.

**Examples**

```r
f <- function(x) {
    z <- 100
    ~ x + z
}
z <- 10
lazy_eval(f(10))
```
**make_call**

Description

In order to exactly replay the original call, the environment must be the same for all of the dots. This function circumvents that a little, falling back to the baseenv() if all environments aren’t the same.

Usage

make_call(fun, args)

Arguments

fun Function as symbol or quoted call.
args Arguments to function; must be a lazy_dots object, or something as.lazy_dots() can coerce.

Value

A list:

env The common environment for all elements
expr The expression

Examples

make_call(quote(f), lazy_dots(x = 1, 2))
make_call(quote(f), list(x = 1, y = ~x))
make_call(quote(f), ~x)

# If no known or no common environment, fails back to baseenv()
make_call(quote(f), quote(x))

```make_call
lazy_eval(f(10), list(x = 100))
lazy_eval(f(10), list(x = 1, z = 1))
lazy_eval(lazy_dots(a = x, b = z), list(x = 10))
```
missing_arg

Generate a missing argument.

Description

Generate a missing argument.

Usage

missing_arg()

Examples

f_interp(-f(x = uq(missing_arg())))
f_interp(-f(x = uq(NULL)))
Index

alist, 7
as.lazy, 2
as.lazy_dots, 19
as.lazy_dots(as.lazy), 2
as_call, 16
as_call(as_name), 3
as_f_list(f_list), 10
as_name, 3, 16
ast(ast_), 3
ast_, 3

baseenv, 19

call_modify, 4
call_new, 5
call_standardise, 4
call_standardise(call_modify), 4
dots_capture(f_capture), 7

dot_env(expr_label), 5
dot_find(expr_label), 5

dot_text(expr_label), 5

dot_text(expr_label), 5

dot_captur(f_captur), 7

f_capture, 7
f_env(f_rhs), 12
f_env<- (f_rhs), 12
f_eval(f_eval_rhs), 8
f_eval_lhs (f_eval_rhs), 8
f_eval_rhs, 8
f_interp, 8, 9
f_label (f_text), 13
f_lhs(f_rhs), 12
f_lhs<- (f_rhs), 12
f_list, 10
f_new, 11
f_rhs, 12
f_rhs<- (f_rhs), 12
f_text, 13

f_unwrap, 13
find_data (f_eval_rhs), 8

function_new, 6

interp, 14
is_atomic(is_lang), 15
is_call(is_lang), 15
is_formula, 15
is_lang, 15
is_name(is_lang), 15

is_pairlist(is_lang), 15

lapply, 2
lazy, 14, 17
lazy (lazy_), 16
lazy_, 16
lazy_dots, 17
lazy_eval, 18

make_call, 19
missing_arg, 20

quote, 7

substitute, 16

uq(f interp), 9
uqf(f interp), 9
uqs(f interp), 9