Package 'pryr'

February 18, 2018

Version 0.1.4

Title Tools for Computing on the Language

Description Useful tools to pry back the covers of R and understand the language at a deeper level.

License GPL-2

URL https://github.com/hadley/pryr

BugReports https://github.com/hadley/pryr/issues

LinkingTo Rcpp

Depends R (>= 3.1.0)

Imports stringr, codetools, methods, Rcpp (>= 0.11.0)

Suggests testthat (>= 0.8.0)

RoxygenNote 6.0.1

NeedsCompilation yes

Author Hadley Wickham [aut, cre],
R Core team [ctb] (Some code extracted from base R)

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2018-02-18 04:42:23 UTC

R topics documented:

`bytes` *Print the byte-wise representation of a value*

Description

Print the byte-wise representation of a value

Usage

```
bytes(x, split = TRUE)

bits(x, split = TRUE)
```

Arguments

x	An R vector of type integer, numeric, logical or character.
split	Whether we should split the output string at each byte.

References

http://en.wikipedia.org/wiki/Two's_complement for more information on the representation used for ints.

http://en.wikipedia.org/wiki/IEEE_floating_point for more information the floating-point representation used for doubles.

http://en.wikipedia.org/wiki/Character_encoding for an introduction to character encoding, and ?Encoding for more information on how R handles character encoding.

Examples

```
## Encoding doesn't change the internal bytes used to represent characters;
## it just changes how they are interpretted!

x <- y <- z <- "\u9b3c"
Encoding(y) <- "bytes"
Encoding(z) <- "latin1"
print(x); print(y); print(z)
bytes(x); bytes(y); bytes(z)
bits(x); bits(y); bits(z)

## In R, integers are signed ints. The first bit indicates the sign, but
## values are stored in a two's complement representation. We see that
## NA_integer_ is really just the smallest negative integer that can be
## stored in 4 bytes
bits(NA_integer_)

## There are multiple kinds of NAs, NaNs for real numbers
## (at least, on 64bit architectures)
print( c(NA_real_, NA_real_ + 1) )
rbind( bytes(NA_real_), bytes(NA_real_ + 1) )
rbind( bytes(NaN), bytes(0/0) )
```

---

call_tree *Display a call (or expression) as a tree.*

Description

call_tree takes a quoted expression. ast does the quoting for you.

Usage

```
call_tree(x, width = getOption("width"))

ast(x)
```

Arguments

x	quoted call, list of calls, or expression to display
width	displays width, defaults to current width as reported by `getOption("width")`

Examples

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

ast(f(x, 1, g(), h(i())))
ast(if (TRUE) 3 else 4)
ast(function(a = 1, b = 2) {a + b})
ast(f()()())
```

---

compose *Compose multiple functions*

Description

In infix and prefix forms.

Usage

```
compose(...)

f %.% g
```

Arguments

...	n functions to apply in order from right to left
f, g	two functions to compose for the infix form

Examples

```
not_null <- `!` %.% is.null
not_null(4)
not_null(NULL)

add1 <- function(x) x + 1
compose(add1,add1)(8)
```

dots *Capture unevaluated dots.*

Description

Capture unevaluated dots.

Usage

```
dots(...)

named_dots(...)
```

Arguments

...	... passed in to the parent function

Value

a list of expressions (not expression objects). named_dots will use the deparsed expressions as default names.

Examples

```
y <- 2
str(dots(x = 1, y, z = ))
str(named_dots(x = 1, y, z =))
```

enclosing_env *Find the environment that encloses of a function.*

Description

This is a wrapper around environment with a consistent syntax.

Usage

```
enclosing_env(f)
```

Arguments

f	The name of a function.

Examples

```
enclosing_env("plot")
enclosing_env("t.test")
```

explicit *Tools for making promises explicit*

Description

Deprecated: please use the lazyeval package instead.

Usage

```
explicit(x)

eval2(x, data = NULL, env = parent.frame())
```

Arguments

x	expression to make explicit, or to evaluate.
data	Data in which to evaluate code
env	Enclosing environment to use if data is a list or data frame.

f *A compact syntax for anonymous functions.*

Description

A compact syntax for anonymous functions.

Usage

```
f(..., .env = parent.frame())
```

Arguments

...	The last argument is the body of the function, all others are arguments to the function. If there is only one argument, the formals are guessed from the code.
.env	parent environment of the created function

Value

a function

Examples

```
f(x + y)
f(x + y)(1, 10)
f(x, y = 2, x + y)

f({y <- runif(1); x + y})
```

fget *Find a function with specified name.*

Description

Find a function with specified name.

Usage

```
fget(name, env = parent.frame())
```

Arguments

name	length one character vector giving name
env	environment to start search in.

Examples

```
c <- 10
fget("c")
```

find_funs *Find functions matching criteria.*

Description

This is a flexible function that matches function component against a regular expression, returning the name of the function if there are any matches. `fun_args` and `fun_calls` are helper functions that make it possible to search for functions with specified argument names, or which call certain functions.

Usage

```
find_funs(env = parent.frame(), extract, pattern, ...)

fun_calls(f)

fun_args(f)

fun_body(f)
```

Arguments

env	environment in which to search for functions
extract	component of function to extract. Should be a function that takes a function as input as returns a character vector as output, like fun_calls or fun_args.
pattern	**stringr** regular expression to results of extract function.
...	other arguments passed on to grepl
f	function to extract information from

Examples

```
find_funs("package:base", fun_calls, "match.fun", fixed = TRUE)
find_funs("package:stats", fun_args, "^[A-Z]+$")

fun_calls(match.call)
fun_calls(write.csv)

fun_body(write.csv)
find_funs("package:utils", fun_body, "write", fixed = TRUE)
```

---

find_uses *Find all functions in that call supplied functions.*

Description

Find all functions in that call supplied functions.

Usage

```
find_uses(envs, funs, match_any = TRUE)
```

Arguments

envs	Vector of environments to look in. Can be specified by name, position or as environment
funs	Functions to look for
match_any	If TRUE return functions that use any of funs. If FALSE, return functions that use all of funs.

Examples

```
names(find_uses("package:base", "sum"))

envs <- c("package:base", "package:utils", "package:stats")
funs <- c("match.call", "sys.call")
find_uses(envs, funs)
```

ftype *Determine function type.*

Description

This function figures out whether the input function is a regular/primitive/internal function, a internal/S3/S4 generic, or a S3/S4/RC method. This is function is slightly simplified as it's possible for a method from one class to be a generic for another class, but that seems like such a bad idea that hopefully no one has done it.

Usage

```
ftype(f)
```

Arguments

f	unquoted function name

Value

a character of vector of length 1 or 2.

See Also

Other object inspection: otype, sexp_type

Examples

```
ftype(`%in%`)
ftype(sum)
ftype(t.data.frame)
ftype(t.test) # Tricky!
ftype(writeLines)
ftype(unlist)
```

is_active_binding *Active binding info*

Description

Active binding info

Usage

```
is_active_binding(x)
```

Arguments

x	unquoted object name

Examples

```
x <- 10
is_active_binding(x)
x %<a-% runif(1)
is_active_binding(x)
y <- x
is_active_binding(y)
```

---

is_promise *Promise info*

Description

Promise info

Usage

```
is_promise(x)

promise_info(x)
```

Arguments

x	unquoted object name

See Also

Other promise tools: uneval

Examples

```
x <- 10
is_promise(x)
(function(x) is_promise(x))(x = 10)
```

make_call *Make and evaluate calls.*

Description

Make and evaluate calls.

Usage

```
make_call(f, ..., .args = list())

do_call(f, ..., .args = list(), .env = parent.frame())
```

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
.env	Environment in which to evaluate call. Defaults to parent frame.

Examples

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

#' Can supply arguments individual or in a list
make_call(quote(f), a = 1, b = 2)
make_call(quote(f), list(a = 1, b = 2))
```

make_function *Make a function from its components.*

Description

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

Usage

```
make_function(args, body, env = parent.frame())
```

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

```
f <- function(x) x + 3
g <- make_function(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))
```

---

mem_change *Determine change in memory from running code*

Description

Determine change in memory from running code

Usage

```
mem_change(code)
```

Arguments

code	Code to evaluate.

Value

Change in memory (in megabytes) before and after running code.

Examples

```
# Need about 4 mb to store 1 million integers
mem_change(x <- 1:1e6)
# We get that memory back when we delete it
mem_change(rm(x))
```

`mem_used` *How much memory is currently used by R?*

Description

R breaks down memory usage into Vcells (memory used by vectors) and Ncells (memory used by everything else). However, neither this distinction nor the "gc trigger" and "max used" columns are typically important. What we're usually most interested in is the the first column: the total memory used. This function wraps around `gc()` to return the total amount of memory (in megabytes) currently used by R.

Usage

```
mem_used()
```

Value

Megabytes of ram used by R objects.

Examples

```
mem_used()
```

`method_from_call` *Given a function class, find correspoding S4 method*

Description

Given a function class, find correspoding S4 method

Usage

```
method_from_call(call, env = parent.frame())
```

Arguments

`call`	unquoted function call
`env`	environment in which to look for function definition

Examples

```
library(stats4)

# From example(mle)
y <- c(26, 17, 13, 12, 20, 5, 9, 8, 5, 4, 8)
nLL <- function(lambda) -sum(dpois(y, lambda, log = TRUE))
fit <- mle(nLL, start = list(lambda = 5), nobs = length(y))

method_from_call(summary(fit))
method_from_call(coef(fit))
method_from_call(length(fit))
```

`modify_call` *Modify the arguments of a call.*

Description

Modify the arguments of a call.

Usage

```
modify_call(call, new_args)
```

Arguments

`call`	A call to modify. It is first standardised with `standardise_call`.
`new_args`	A named list of expressions (constants, names or calls) used to modify the call. Use `NULL` to remove arguments.

Examples

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

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

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

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

# Add an explicit missing argument
modify_call(call, list(na.rm = quote(expr = )))
```

modify_lang *Recursively modify a language object*

Description

Recursively modify a language object

Usage

```
modify_lang(x, f, ...)
```

Arguments

x	object to modify: should be a call, expression, function or list of the above.
f	function to apply to leaves
...	other arguments passed to f

Examples

```
a_to_b <- function(x) {
  if (is.name(x) && identical(x, quote(a))) return(quote(b))
  x
}
examples <- list(
  quote(a <- 5),
  alist(a = 1, c = a),
  function(a = 1) a * 10,
  expression(a <- 1, a, f(a), f(a = a))
)
modify_lang(examples, a_to_b)
# Modifies all objects called a, but doesn't modify arguments named a
```

object_size *Compute the size of an object.*

Description

object_size works similarly to object.size, but counts more accurately and includes the size of environments. compare_size makes it easy to compare the output of object_size and object.size.

Usage

```
object_size(..., env = parent.frame())

compare_size(x)
```

Arguments

env	Environment in which to terminate search. This defaults to the current environment so that you don't include the size of objects that are already stored elsewhere.
x, ...	Set of objects to compute total size.

Value

An estimate of the size of the object, in bytes.

Environments

object_size attempts to take into account the size of the environments associated with an object. This is particularly important for closures and formulas, since otherwise you may not realise that you've accidentally captured a large object. However, it's easy to over count: you don't want to include the size of every object in every environment leading back to the emptyenv(). object_size takes a heuristic approach: it never counts the size of the global env, the base env, the empty env or any namespace.

Additionally, the env argument allows you to specify another environment at which to stop. This defaults to the environment from which object_size is called to prevent double-counting of objects created elsewhere.

Examples

```
# object.size doesn't keep track of shared elements in an object
# object_size does
x <- 1:1e4
z <- list(x, x, x)
compare_size(z)

# this means that object_size is not transitive
object_size(x)
object_size(z)
object_size(x, z)

# object.size doesn't include the size of environments, which makes
# it easy to miss objects that are carrying around large environments
f <- function() {
  x <- 1:1e4
  a ~ b
}
compare_size(f())
```

otype *Determine object type.*

Description

Determine object type.

Usage

```
otype(x)
```

Arguments

x	object to determine type of

Details

Figure out which object system an object belongs to:

- base: no class attribute
- S3: class attribute, but not S4
- S4: `isS4`, but not RC
- RC: inherits from "refClass"

See Also

Other object inspection: `ftype`, `sexp_type`

Examples

```
otype(data.frame())
otype(1:10)
```

parent_promise *Find the parent (first) promise.*

Description

Find the parent (first) promise.

Usage

```
parent_promise(x)
```

Arguments

x	unquoted name of promise to find initial value for for.

Examples

```
f <- function(x) g(x)
g <- function(y) h(y)
h <- function(z) parent_promise(z)

h(x + 1)
g(x + 1)
f(x + 1)
```

---

parenv *Get parent/ancestor environment*

Description

Get parent/ancestor environment

Usage

```
parenv(env = parent.frame(), n = 1)
```

Arguments

env	an environment
n	number of parents to go up

Examples

```
adder <- function(x) function(y) x + y
add2 <- adder(2)
parenv(add2)
```

---

parenvs *Given an environment or object, return an* envlist *of its parent environments.*

Description

If e is not specified, it will start with environment from which the function was called.

Usage

```
parenvs(e = parent.frame(), all = FALSE)
```

Arguments

e	An environment or other object.
all	If FALSE (the default), stop at the global environment or the empty environment. If TRUE, print all parents, stopping only at the empty environment (which is the top-level environment).

Examples

```
# Print the current environment and its parents
parenvs()

# Print the parent environments of the load_all function
e <- parenvs(parenvs)
e

# Get all parent environments, going all the way to empty env
e <- parenvs(parenvs, TRUE)
e

# Print e with paths
print(e, path = TRUE)

# Print the first 6 environments in the envlist
e[1:6]

# Print just the parent environment of load_all.
# This is an envlist with one element.
e[1]

# Pull that environment out of the envlist and see what's in it.
e[[1]]
ls(e[[1]], all.names = TRUE)
```

---

partial *Partial apply a function, filling in some arguments.*

Description

Partial function application allows you to modify a function by pre-filling some of the arguments. It is particularly useful in conjunction with functionals and other function operators.

Usage

```
partial(`_f`, ..., .env = parent.frame(), .lazy = TRUE)
```

Arguments

_f	a function. For the output source to read well, this should be an be a named function. This argument has the weird (non-syntactic) name _f so it doesn't accidentally capture any argument names begining with f.
...	named arguments to f that should be partially applied.
.env	the environment of the created function. Defaults to parent.frame and you should rarely need to modify this.
.lazy	If TRUE arguments evaluated lazily, if FALSE, evaluated when partial is called.

Design choices

There are many ways to implement partial function application in R. (see e.g. dots in https://github.com/crowding/ptools for another approach.) This implementation is based on creating functions that are as similar as possible to the anonymous function that'd you'd create by hand, if you weren't using partial.

Examples

```
# Partial is designed to replace the use of anonymous functions for
# filling in function arguments. Instead of:
compact1 <- function(x) Filter(Negate(is.null), x)

# we can write:
compact2 <- partial(Filter, Negate(is.null))

# and the generated source code is very similar to what we made by hand
compact1
compact2

# Note that the evaluation occurs "lazily" so that arguments will be
# repeatedly evaluated
f <- partial(runif, n = rpois(1, 5))
f
f()
f()

# You can override this by saying .lazy = FALSE
f <- partial(runif, n = rpois(1, 5), .lazy = FALSE)
f
f()
f()

# This also means that partial works fine with functions that do
# non-standard evaluation
my_long_variable <- 1:10
plot2 <- partial(plot, my_long_variable)
plot2()
plot2(runif(10), type = "l")
```

rebind *Rebind an existing name.*

Description

This function is similar to <<- with two exceptions:

Usage

```
rebind(name, value, env = parent.frame())
```

Arguments

name	name of existing binding to re-assign
value	new value
env	environment to start search in.

Details

- if no existing binding is found, it throws an error
- it does not recurse past the global environment into the attached packages

Examples

```
a <- 1
rebind("a", 2)
a
# Throws error if no existing binding
## Not run: rebind("b", 2)

local({
  rebind("a", 3)
})
a

# Can't find get because doesn't look past globalenv
## Not run: rebind("get", 1)
```

rls *Recursive ls.*

Description

Performs `ls` all the way up to a top-level environment (either the parent of the global environment, the empty environment or a namespace environment).

Usage

```
rls(env = parent.frame(), all.names = TRUE)
```

Arguments

`env`	environment to start the search at. Defaults to the `parent.frame`. If a function is supplied, uses the environment associated with the function.
`all.names`	Show all names, even those starting with `.`? Defaults to `TRUE`, the opposite of `ls`

Author(s)

Winston Chang

sexp_type *Inspect internal attributes of R objects.*

Description

`typename` determines the internal C typename, `address` returns the memory location of the object, and `refs` returns the number of references pointing to the underlying object.

Usage

```
sexp_type(x)

inspect(x, env = parent.frame())

refs(x)

address(x)

typename(x)
```

Arguments

x	name of object to inspect. This can not be a value.
env	When inspecting environments, don't go past this one.

Non-standard evaluation

All functions uses non-standard evaluation to capture the symbol you are referring to and the environment in which it lives. This means that you can not call any of these functions on objects created in the function call. All the underlying C level functions use `Rf_findVar` to get to the underlying SEXP.

See Also

Other object inspection: `ftype`, `otype`

Examples

```
x <- 1:10
## Not run: .Internal(inspect(x))

typename(x)
refs(x)
address(x)

y <- 1L
typename(y)
z <- list(1:10)
typename(z)
delayedAssign("a", 1 + 2)
typename(a)
a
typename(a)

x <- 1:5
address(x)
x[1] <- 3L
address(x)
```

---

show_c_source *Find C source code for internal R functions*

Description

Opens a link to code search on github.

Usage

```
show_c_source(fun)
```

Arguments

fun	.Internal or .Primitive function call.

Examples

```
show_c_source(.Internal(mean(x)))
show_c_source(.Primitive(sum(x)))
```

---

standardise_call *Standardise a function call*

Description

Standardise a function call

Usage

```
standardise_call(call, env = parent.frame())
```

Arguments

call	A call
env	Environment in which to look up call value.

---

subs *A version of substitute that works in the global environment.*

Description

This version of substitute is more suited for interactive exploration because it will perform substitution in the global environment: the regular version has a special case for the global environment where it effectively works like quote

Usage

```
subs(x, env = parent.frame())
```

Arguments

x	a quoted call
env	an environment, or something that behaves like an environment (like a list or data frame), or a reference to an environment (like a positive integer or name, see as.environment for more details)

Substitution rules

Formally, substitution takes place by examining each name in the expression. If the name refers to:

- an ordinary variable, it's replaced by the value of the variable.
- a promise, it's replaced by the expression associated with the promise.
- . . . , it's replaced by the contents of . . .

Examples

```
a <- 1
b <- 2

substitute(a + b)
subs(a + b)
```

---

substitute_q *A version of substitute that evaluates its first argument.*

Description

This version of substitute is needed because `substitute` does not evaluate it's first argument, and it's often useful to be able to modify a quoted call.

Usage

```
substitute_q(x, env)
```

Arguments

`x`	a quoted call
`env`	an environment, or something that behaves like an environment (like a list or data frame), or a reference to an environment (like a positive integer or name, see `as.environment` for more details)

Examples

```
x <- quote(a + b)
substitute(x, list(a = 1, b = 2))
substitute_q(x, list(a = 1, b = 2))
```

track_copy *Track if an object is copied*

Description

The title is somewhat misleading: rather than checking if an object is modified, this really checks to see if a name points to the same object.

Usage

```
track_copy(var, env = parent.frame(), quiet = FALSE)
```

Arguments

var	variable name (unquoted)
env	environment name in which to track changes
quiet	if FALSE, prints a message on change; if FALSE only the return value of the function is used

Value

a zero-arg function, that when called returns a boolean indicating if the object has changed since the last time this function was called

Examples

```
a <- 1:5
track_a <- track_copy(a)
track_a()
a[3] <- 3L
track_a()
a[3] <- 3
track_a()
rm(a)
track_a()
```

unenclose *Unenclose a closure.*

Description

Unenclose a closure by substituting names for values found in the enclosing environment.

Usage

```
unenclose(f)
```

Arguments

f	a closure

Examples

```
power <- function(exp) {
  function(x) x ^ exp
}
square <- power(2)
cube <- power(3)

square
cube
unenclose(square)
unenclose(cube)
```

---

uneval *Capture the call associated with a promise.*

Description

This is an alternative to subsitute that performs one job, and so gives a stronger signal regarding the intention of your code. It returns an error if the name is not associated with a promise.

Usage

```
uneval(x)
```

Arguments

x	unquoted variable name that refers to a promise. An error will be thrown if it's not a promise.

See Also

Other promise tools: `is_promise`

Examples

```
f <- function(x) {
   uneval(x)
}
f(a + b)
f(1 + 4)

delayedAssign("x", 1 + 4)
uneval(x)
x
uneval(x)
```

where *Find where a name is defined.*

Description

Implements the regular scoping rules, but instead of returning the value associated with a name, it returns the environment in which it is located.

Usage

```
where(name, env = parent.frame())
```

Arguments

name	name, as string, to look for
env	environment to start at. Defaults to the calling environment of this function.

Examples

```
x <- 1
where("x")
where("t.test")
where("mean")
where("where")
```

%<a-% *Create an active binding.*

Description

Infix form of makeActiveBinding which creates an *active* binding between a name and an expression: every time the name is accessed the expression is recomputed.

Usage

```
x %<a-% value
```

Arguments

x	unquoted expression naming variable to create
value	unquoted expression to evaluate every time name is accessed

Examples

```
x %<a-% runif(1)
x
x
x %<a-% runif(10)
x
x
rm(x)
```

`%<c-%` *Create a constant (locked) binding.*

Description

Infix wrapper for `assign` + `lockBinding` that creates a constant: a binding whose value can not be changed.

Usage

```
x %<c-% value
```

Arguments

`x`	unquoted expression naming variable to create
`value`	constant value

Examples

```
x %<c-% 10
#' Generates an error:
## Not run: x <- 20

# Note that because of R's operator precedence rules, you
# need to wrap compound RHS expressions in ()
y %<c-% 1 + 2
y
z %<c-% (1 + 2)
z
```

`%<d-%` *Create an delayed binding.*

Description

Infix form of `delayedAssign` which creates an *delayed* or lazy binding, which only evaluates the expression the first time it is used.

Usage

```
x %<d-% value
```

Arguments

`x`	unquoted expression naming variable to create
`value`	unquoted expression to evaluate the first time `name` is accessed

Examples

```
x %<d-% (a + b)
a <- 10
b <- 100
x
```

Index