Controlling evaluation

Hadley Wickham

Assistant Professor / Dobelman Family Junior Chair

Department of Statistics / Rice University

June 2011

1. Motivation

2. Controlling evaluation

3. Practice on other functions

4. Computing on the language

https://github.com/hadley/devtools/wiki/Evaluation

Motivation

```r
subset(mtcars, cyl == 4)
# vs.
mtcars[mtcars$cyl == 4, ]


# How does it work?
subset <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}
```

Motivation

Our chief motivation is to create functions that reduce typing, which is particularly important for interactive use. One way of creating *domain specific languages*.

But note this comes at a cost of making programming with these functions harder.

1. Capture what you typed without evaluating it (**quoting**)

2. Look for the variables in the right place (**evaluating**)

3. Deal with a few special cases (**scoping**)

Quoting


Ceci n'est pas une pipe.

A **symbol**, or **name**, is the name of an object, like x or y, not its value, like 5 or "a".

A **call** is a function call, like sum(1:10) or x == y, not the result of that function.

An **expression** is a list of calls and symbols, like the body of a function.

A **language object** is any of the above or a constant (number or string).

NB: these are my definitions - there is not a lot of consistency in the documentation or amongst R core.

```
x <- quote(vs == am)
x
# vs == am
str(x)
# language vs == am

is.language(x)
# [1] TRUE
is.call(x)
# [1] TRUE
is.expression(x)
# [1] FALSE
```

```
call("==", "vs", "am")
call("==", vs, am)
call("==", as.name("vs"), as.name("am"))
```

```
x <- parse(text = "vs == am")
is.expression(x)
# [1] TRUE

# Expressions are a list of calls/symbols
x[[1]]
# vs == am
is.call(x[[1]])
# [1] TRUE
```

Your turn

Why won't this work?

```
subset <- function(x, condition) {
  quote(condition)
}
subset(mtcars, cyl == 4)
```

```
subset <- function(x, condition) {
  match.call()
}
subset(mtcars, vs == am)
# subset(x = mtcars, condition = vs == am)
```

```
subset <- function(x, condition) {
  match.call()$condition
}
subset(mtcars, vs == am)
# subset(x = mtcars, condition = vs == am)
```

```
subset <- function(x, condition) {
  substitute(condition)
}
subset(mtcars, vs == am)
```

```
# Uses lazy evaluation and extracts
# call from promise/thunk

# Also has other uses
```

quote: captures call without evaluating it

call: builds up a call from component pieces

parse: converts text representing a call into a call

match.call: captures how a function was called

substitute: uses lazy evaluation to extract the call

Evaluating

Evaluation

Now we've captured the condition call, we want to evaluate it in the context of a data frame: instead of looking up the symbols in the global environment, we want to look them up in a data frame

Environments

An **environment** is a list of symbols and their values. Every environment (apart from the base environment) also has a **parent**.

This is same idea as a list or data frame. (Except that they don't have parents)

```
# Given a call and an environment (or something like
# an environment like a list or data frame), eval
# will evaluate the call in that environment

x <- quote(vs == am)
eval(x, globalenv())
eval(x, mtcars)

# What will happen when I run this code?
eval(vs == am, mtcars)
```

```r
subset <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x)
  x[r, ]
}

subset(mtcars, cyl == 4)

# It works!
```

Scoping

```r
# What should this do?

x <- 4
subset(mtcars, cyl == x)

y <- 4
subset(mtcars, cyl == y)

# What does it do?
# Why?
```

```
# We need to tell eval where to look if the
# variables aren't found in the data frame.
# We need to provide the equivalent of a parent
# environment.  That's the third argument to eval

subset <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

# parent.frame() finds the environment in which
# the current function is being executed
```

```
x <- 4
f1 <- function() {
  x <- 6
  subset(mtcars, cyl == x)
}
f1()
```

```
f2 <- function() {
  x <- 8
  subset(mtcars, cyl == get("x"))
}
f2()
```

```r
# An alternative approach would be to use a formula
# Formulas quote and capture the environment in
# which they are defined

subset <- function(x, f) {
  r <- eval(f[[2]], x, environment(f))
  x[r, ]
}
subset(mtcars, ~ cyl == x)

# Using formulas has the advantage that it's
# very obvious that something non-standard is
# going on
```

Other functions

Package	Function
base	with
base	transform
plyr	mutate
plyr	arrange
plyr	summarise

Your turn

Look at `subset.data.frame`. How does it differ to our version? (Consult the documentation if you're not familiar with all the parameters)

Look at `transform.data.frame`. What does it do? How does it work? Why is the first argument called `` `_data` ``?

```
# All these functions are useful for interactive
# data analysis, but ARE NOT suitable for
# programming with.

scramble <- function(x) x[sample(nrow(x)), ]
scramble(mtcars)
```

```
subscramble <- function(x, condition) {
  scramble(subset(x, condition))
}
subscramble(mtcars, cyl == 4)
debugonce(subset)
subscramble(mtcars, cyl == 4)
```

Standard nonstandard evaluation rules

Thomas Lumley

March 19, 2003

This document is designed to clarify the various evaluation rules for function arguments in R and to make some suggestions for new code. The descriptions are based on R 1.5.1.

1 Standard evaluation model

R passes arguments by value: the arguments are evaluated in the calling environment and their values are passed to the function. If arguments are not specified then defaults are used and these are evaluated in the environment inside the function, so that local variables are found first, and then variables visible in the environment where the function was defined.

The evaluation of defaults in the environment inside the function is important, but can be abused. In my opinion we should discourage

```
function (formula, data = parent.frame(), ..., subset, ylab = varnames[response],
    ask = TRUE)
```

Computing on the language

```
# How can we call a function that uses non-standard
# evaluation?

library(lattice)
xyplot(disp ~ mpg, data = mtcars)

x <- "disp"
y <- "mpg"
xyplot(x ~ y, data = mtcars)
```

```
# Second use of substitute: modifying calls
# Extremely useful when, for whatever reason, you
# need to create a call as if you had typed that
# code directly into the command line

substitute(x ~ y, list("x" = x, "y" = y))
```

```
substitute(x ~ y,
  list("x" = as.name(x), "y" = as.name(y)))

eval(substitute(x ~ y,
  list("x" = as.name(x), "y" = as.name(y))))
```

```
f <- substitute(x ~ y, list(x = as.name(x),
  y = as.name(y)))
xyplot(f, data = mtcars)

f <- eval(substitute(x ~ y, list(x = as.name(x),
  y = as.name(y))))
xyplot(f, data = mtcars)

eval(substitute(xyplot(x ~ y, data = mtcars),
  list(x = as.name(x), y = as.name(y))))
```

Your turn

Rewrite `subscramble` using `substitute` and `eval` so that it works.

```
subscramble <- function(x, condition) {
  condition_call <- substitute(condition)
  eval(substitute(scramble(subset(x, condition)),
      list(condition = condition_call)))
}
```

```
# calls are trees, and behave like lists
x <- quote(a * (b + 1))

# First piece is name of function being called
x[[1]]

# Subsequent pieces are arguments (language objects)
as.list(x[-1])
x[[2]]
x[[3]]
x[[3]][[1]]
x[[3]][[2]]
x[[3]][[2]][[1]]
```

```
# can modify calls
x <- quote(a * (b + 1))

x[[1]] <- as.name("c")
x

x[[1]] <- as.name("*")
x
```

```
y <- quote(lm(formula = disp ~ mpg, data = mtcars))
y$formula <- quote(price ~ carat)
y$data <- quote(diamonds)

# See 2-draw-tree.r for an example that
# draws call trees in a more informative manner.
```

Your turn

Read the code for `write.csv`. How does it work? How could you rewrite it more simply?

What are the advantages/disadvantages of the current and the simpler approaches?

```
write.csv <- function (x, file = "", quote = TRUE,
  eol = "\n", na = "NA", row.names = TRUE,
  fileEncoding = "") {

  write.table(x, file, quote = quote, eol = eol,
    na = na, row.names = row.names,
    fileEncoding = fileEncoding, sep = ",",
    dec = ".", qmethod = "double")
}

# Main disadvantage is that you need to update the
# arguments to write.csv if write.table changes
```

Conclusions

Conclusions

Subset illustrates many important foundational R ideas: quoting, evaluating and scoping.

Mastering these techniques allows you to access a higher level of abstraction and can make many previously difficult problems easier to solve.

But expect frustration! It is not intuitive


Ceci n'est pas une pipe.