Metaprogramming

Hadley Wickham

Chief Scientist

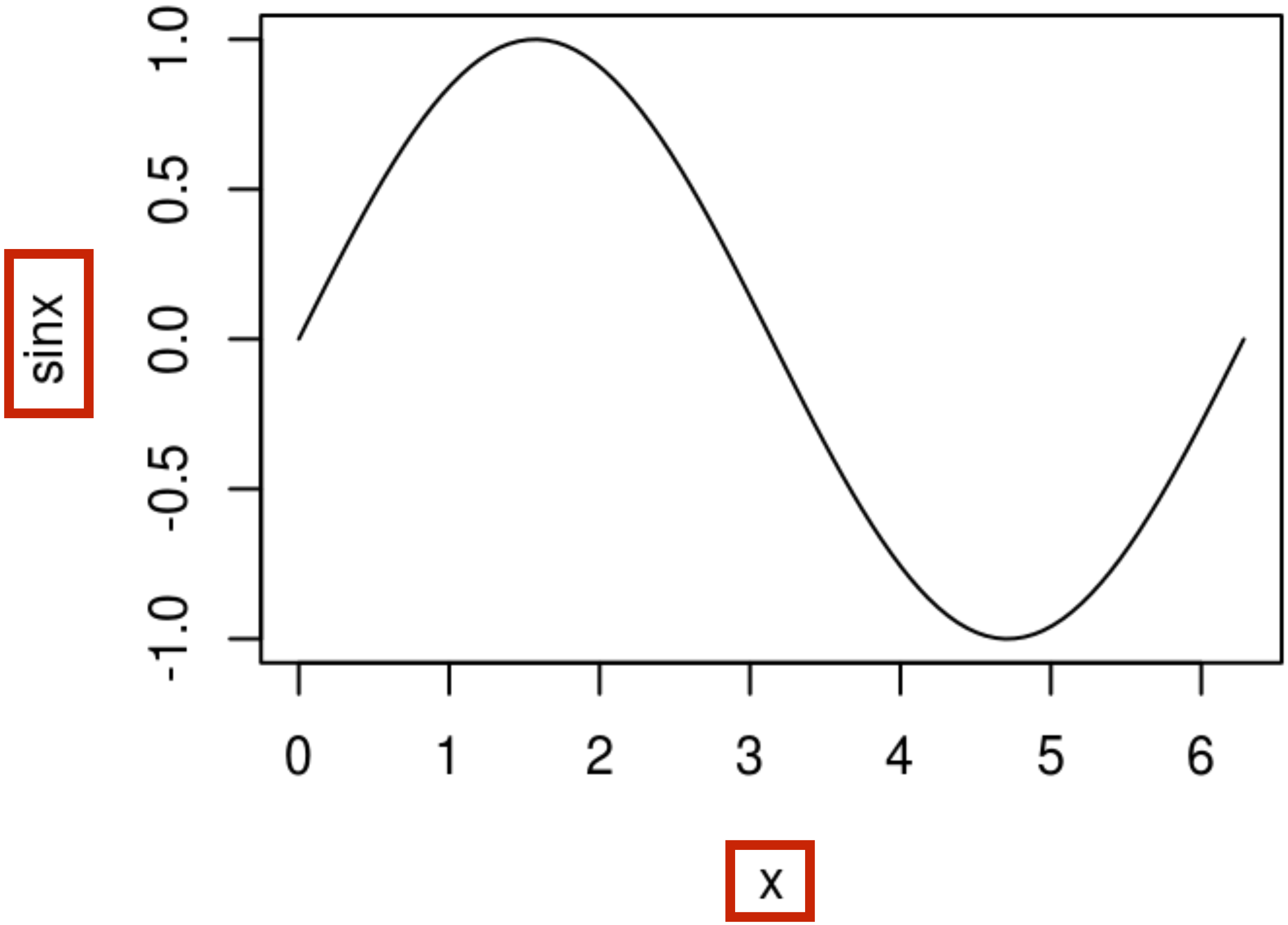
RStudio

May 2015

1. Motivation
2. Warmups
3. Calls
4. Subset
5. Escape hatches
6. Where next

Motivation

```
x <- seq(0, 2 * pi, length = 100)
sinx <- sin(x)
plot(x, sinx, type = ”l”)
```



```
# What does this do?

library(plyr)

plyr <- ”ggplot2”
library(plyr)

library(plyr, character.only = TRUE)
```

```r
lm(mpg ~ wt, data = mtcars)
mpg ~ wt
```

```
# What we’re going to explore today

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

Motivation

Many key pieces of R using metaprogramming and non-standard evaluation.

For better or worse, metaprogramming magic is a key part of R’s magic, so it’s a good idea to understand the basics.

Warmups

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

—John Chambers

```r
# In most R code, the function name comes first
f(10, g(12, 3))

# But sometimes it’s in the middle
10 - (12 + 3)

# In R, you can always convert the infix to prefix
`-`(10, `+`(12, 3))

# And EVERY SINGLE OPERATION in R gets converted
# to this form internally
# That’s why you can do crazy stuff like this:

`(` <- function(e1) {
  if (is.numeric(e1) && runif(1) < 0.1) {
    e1 + 1
  } else {
    e1
  }
}
replicate(50, (1 + 2))
#>  [1] 3 3 3 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4
#> [22] 3 4 3 3 3 3 4 4 4 3 3 3 3 3 3 3 3 3 4 3 3
#> [43] 3 3 3 3 3 3 3 3

rm(”(”)
```

Your turn

Translate these calls into their regular form

```
`if`(x > 10, ”a”, ”b”)

`*`(`(`(`+`(1, 2)), 3)
```

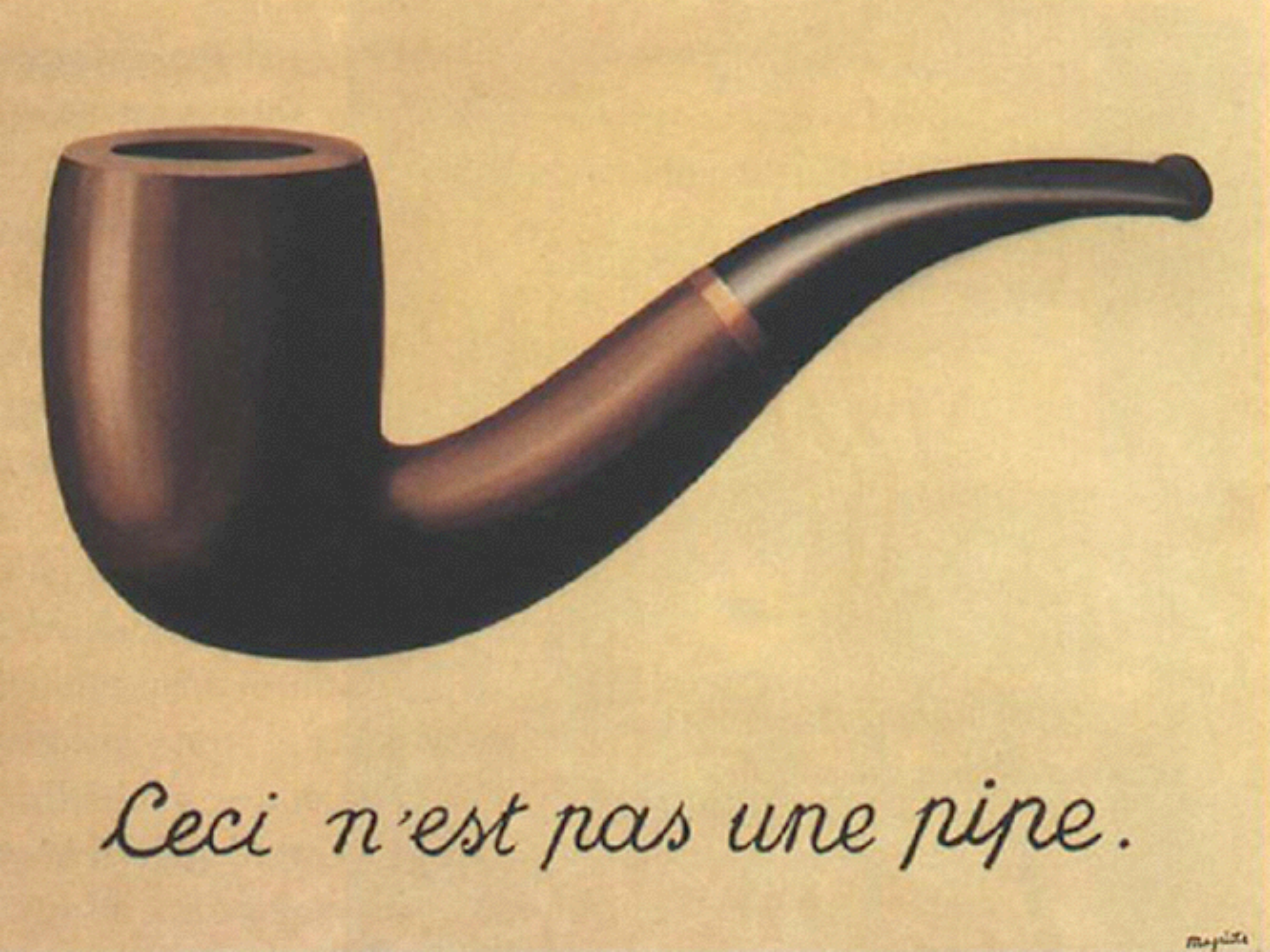
```r
# Your turn: what does the following code do?

# Translate it into basic subsetting &
# assignment. What's special about subset and
# transform?

subset(mtcars, cyl == 4)
subset(mtcars, select = disp:wt)

transform(mtcars, cyl2 = cyl * 2)
transform(mtcars, mpg = NULL)
```

Calls


Ceci n'est pas une pipe.
Magritte

```
# Need to be able to separate code from its
# actions. quote() allows us to capture the
# code for an action
quote(mean(1:10, na.rm = TRUE))
quote(library(ggplot2))
quote(a + (b * c))

# It also captures names
quote(a)

# And constants
quote(12345)
quote(”a”)
```

~~`expression()`~~

Returns complicated object

```
library(pryr)
# A function call is like a tree. The leaves
# are names (aka symbols) and constants.

ast(mean(1:10, na.rm = TRUE))
ast(a + (b * c))
ast(if(TRUE) 1 else stop(”!”))
ast(f(1)(5))

ast(”a”)
ast(x)

# Why can a leaf never be a vector of length
# greater than one?
```

```r
# Lists are also like trees, and you can
# subset calls like you can subset lists

call <- quote(mean(1:10, na.rm = TRUE))

# The first element is the function
call[[1]]

# The subsequent elements are the arguments
call[[2]]
call[[3]]
call$na.rm
```

```
# The opposite of quote() is eval. It takes
# call, name or constant, and evaluates it

eval(quote(mean(1:10, na.rm = TRUE)))
eval(quote(library(ggplot2)))
```

```r
# Your turn: predict the results of the following
# code before running it (remember that eval
# cancels a quote)

eval(quote(eval(quote(eval(quote(2 + 2))))))
eval(eval(quote(eval(quote(eval(quote(2 + 2)))))))
quote(eval(quote(eval(quote(eval(quote(2 + 2)))))))
```

Subset

Goal

- Implement our own version of `subset()`, `subset2()`
- Understand what every line of code does!
- Three steps: capture desired action, evaluate in right environment, subset data frame

Step 1

Why won’t this work? What does it always return?

```
capture_x <- function(x) {
  quote(x)
}
capture_x(cyl == 4)
```

Brainstorm with your neighbours for one minute.

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

# How does it work? R uses lazy evaluation, so
# every argument is not a value but a promise to
# compute a value. The promise contains a quoted
# call and an environment. First use of
# substitute() is extract a quoted call from a
# promise.
```

Evaluation

Now we’ve captured the condition as a 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 names and associated values. Every environment (apart from the empty 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) # seen this already
eval(x, globalenv()) # more explicit
eval(x, mtcars) # looks first in mtcars

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

```r
subset2 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x)
  x[r, ]
}

subset2(mtcars, cyl == 4)

# It works!
# Or does it??
y <- 4
subset(mtcars, cyl == y)

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

# What does it do? Why?
# Hint: why are x and y different?
# 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
```

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

Package	Function
base	`with()`
base	`transform()`
plyr	`mutate()`
plyr	`arrange()`
plyr	`summarise()`

dplyr functions better for data analysis, but all code in C++

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` ``?

Escape hatches

```r
# 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)
```

Escape hatches

- Every function that uses non-standard evaluation should provide a standard-evaluation (SE) equivalent
- SE equivalent should end in _

```
subset_ <- function(x, cond, env) {
  r <- eval(cond, x, env)
  x[r, ]
}
subset <- function(x, cond) {
  subset_(x, substitute(cond), parent.frame()
}

subscramble <- function(x, cond) {
  subs <- subset_(x, substitute(cond),
    parent.frame())
  scramble(sub)
}
```

```
# What if the function doesn’t provide an
# escape hatch?

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

x <- ”disp”
y <- ”mpg”
xyplot(y ~ x, 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(f(y))
substitute(f(y), list(y = 1))
substitute(f(y), list(f = ”g”))
substitute(f(y), list(f = quote(g)))

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

```r
# Can use as.name to turn string into a name of
# a variable
vars <- list(x = as.name(x), y = as.name(y))
substitute(x ~ y, vars)

# What does substitute return? Will this work?
f <- substitute(x ~ y, vars)
xyplot(f, data = mtcars)
```

```
# substitute() returns a quoted call -
# to evaluate it to get a formula
f <- eval(substitute(x ~ y, vars))
xyplot(f, data = mtcars)

# Or work with the complete call
eval(substitute(xyplot(x ~ y, data = mtcars), vars))
```

Your turn

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

(Hint: you'll need to construct a call to `subset()` and then evaluate it)

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

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

subscramble(mtcars, cyl == 4)
# This is complicated!
```

Beware

`write.csv()` uses it to construct a call to `write.table()` – really bad idea

```
write.csv <- function(...) {
  Call <- match.call(expand.dots = TRUE)
  rn <- eval.parent(Call$row.names)
  Call$append <- NULL
  Call$col.names <- if(is.logical(rn) && !rn) TRUE else NA
  Call$sep <- ”,”
  Call$dec <- ”.”
  Call$qmethod <- ”double”
  Call[[1L]] <- quote(write.table)
  eval(Call, parent.frame())
}
```

```
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
```

Where next

http://adv-r.had.co.nz/Environments.html

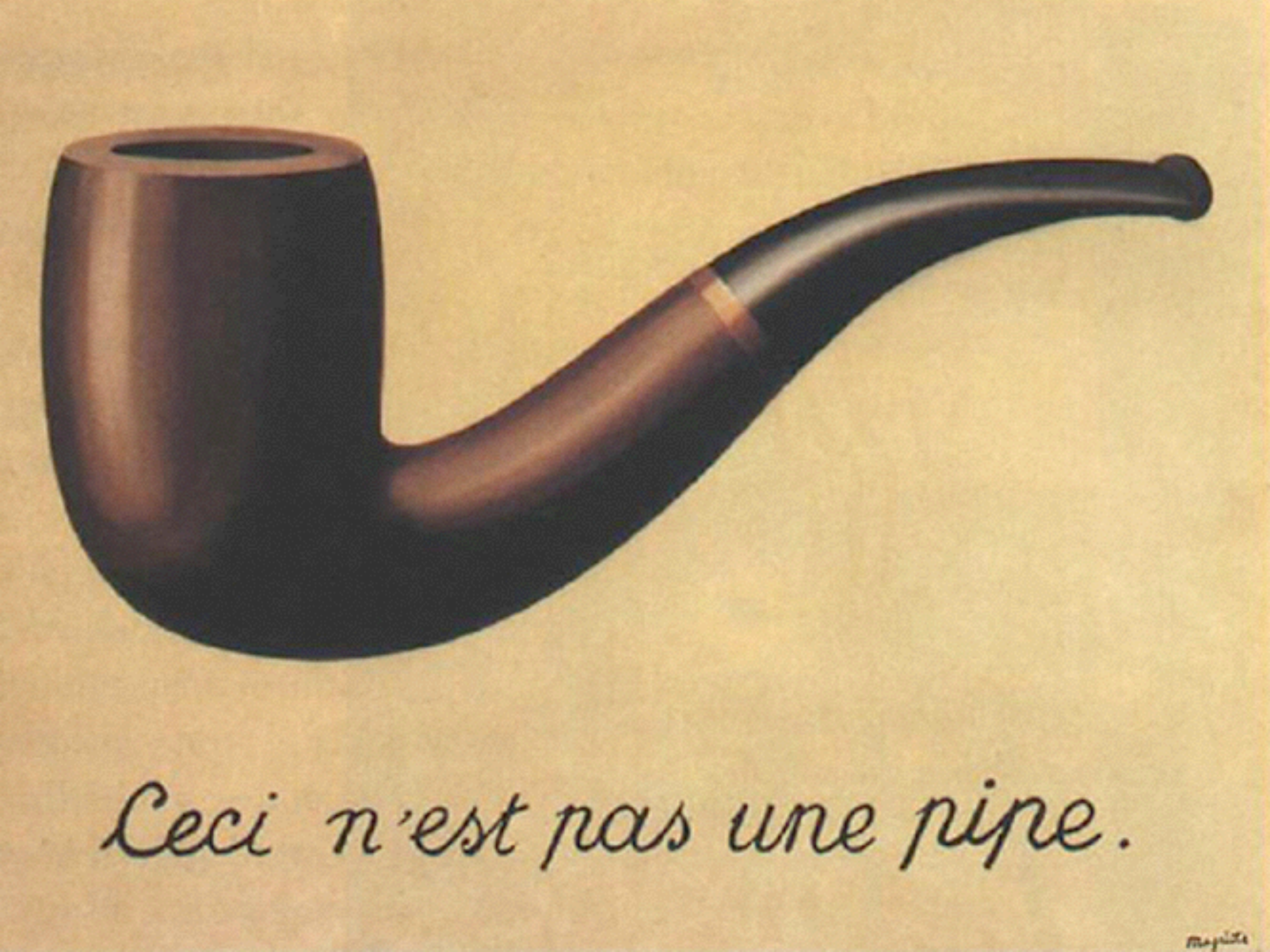
http://adv-r.had.co.nz/Computing-on-the-language.html

http://adv-r.had.co.nz/Expressions.html

http://adv-r.had.co.nz/dsl.html

```
# General tools and procedures in
```

https://github.com/hadley/lazyeval



Ceci n'est pas une pipe.
Magritte