# lazyeval

## A uniform approach to NSE

***July 2016***

Hadley Wickham
@hadleywickham
Chief Scientist, **RStudio**

# Motivation

# Take this simple variant of subset()

```
subset <- function(df, condition) {
  cond <- substitute(condition)
  rows <- eval(cond, df, parent.frame())

  rows[is.na(rows)] <- FALSE
  df[rows, , drop = FALSE]
}
```

# Pro: it reduces typing

```
subset(
  my_data_frame_with_a_very_long_name,
  x > 10 & y > 10
)

# vs.

my_data_frame_with_a_very_long_name[
  my_data_frame_with_a_very_long_name$x > 10 &
  my_data_frame_with_a_very_long_name$y > 10,
]

# and hence makes the code clearer
```

# Pro: it alleviates two common frustrations

```
df <- data.frame(x = c(1:5, NA))
subset(df, x > 3)
#>   x
#> 4 4
#> 5 5

# vs.

df[df$x > 3, ]
#> [1]  4  5 NA
```

# **Con:** you can't define then use the arguments

```
rows <- cyl == 6
my_subset(mtcars, row)
```

# Con: it fails with the simplest wrapper

```
my_subset <- function(df, cond) {
  subset(df, cond)
}
my_subset(mtcars, cyl == 6)
#> Error in eval(expr, envir, enclos) :
#>   object 'cyl' not found
```

# Con: it’s hard to safely compose

```r
threshold_x <- function(df, threshold) {
  subset(df, x > threshold)
}

# Silently gives incorrect result if:
# (a) no x col in df, but x var in parent
# (b) df has threshold column
```

# Con: it’s hard to safely parameterise

```
# I think this is the best you can do
threshold <- function(df, var, threshold) {
  stopifnot(is.name(var))

  eval(substitute(subset(df, var > threshold)))
}
```

Can we do better?

# Can we do better?

```
subset <- function(df, condition) {
  cond <- substitute(condition)
  rows <- eval(cond, df, parent.frame())

  rows[is.na(rows)] <- FALSE
  df[rows, , drop = FALSE]
}
```

# Here is one approach

```
sieve <- function(df, condition) {
  rows <- lazyeval::f_eval(condition, df)

  rows[is.na(rows)] <- FALSE
  df[rows, , drop = FALSE]
}
```

# **Con:** requires 1-2 more characters

```
subset(mtcars, mpg > 30)

# vs.

sieve(mtcars, ~ mpg > 30)
```

# Pro: it’s referentially transparent

```
# This works:
x <- ~ mpg > 30
sieve(mtcars, x)

# As does this:
my_sieve <- function(df, condition) {
  sieve(df, condition)
}

# And this:
n <- 10
my_sieve(mtcars, ~ x > n)
```

# Why does this work?

```
library(lazyeval)

# Because a formula captures both the
# expression and the environment

f <- ~ mpg > 30

f_rhs(f)
#> mpg > 30
f_env(f)
#> <environment: R_GlobalEnv>
```

# Most important new function is f_eval()

```
sieve <- function(df, condition) {
  rows <- f_eval(condition, df)

  rows[is.na(rows)] <- FALSE
  df[rows, , drop = FALSE]
}
```

# f_eval() is mostly simple:

```
# f_eval() is 90% this:
f_eval <- function(f, data) {
  eval(f_rhs(f), data, f_env(f))
}

# But it provides two useful features:
# (a) pronouns to disambiguate
# (b) full quasiquotation engine
```

# Can use pronouns in to disambiguate:

```
threshold_x <- function(df, threshold) {
  sieve(df, ~ .data$x > .env$threshold)
}

# This will never fail silently
```

# Can use quasiquotation to parameterise:

```
threshold <- function(df, var, threshold) {
  sieve(df, ~ uq(var) > .env$threshold)
}
threshold(mtcars, ~mpg, 30)

# Similar to to bquote() but also provides
# unquote-splice: uqs()
```

# What if you want to eliminate the ~?

Turns promise into formula

```
sieve <- function(df, condition) {
  sieve_(df, f_capture(condition))
}
```

Convention: always provide SE version with _ suffix

```
sieve_ <- function(df, condition) {
  rows <- f_eval(condition, df)
  rows[is.na(rows)] <- FALSE

  df[rows, , drop = FALSE]
}
```

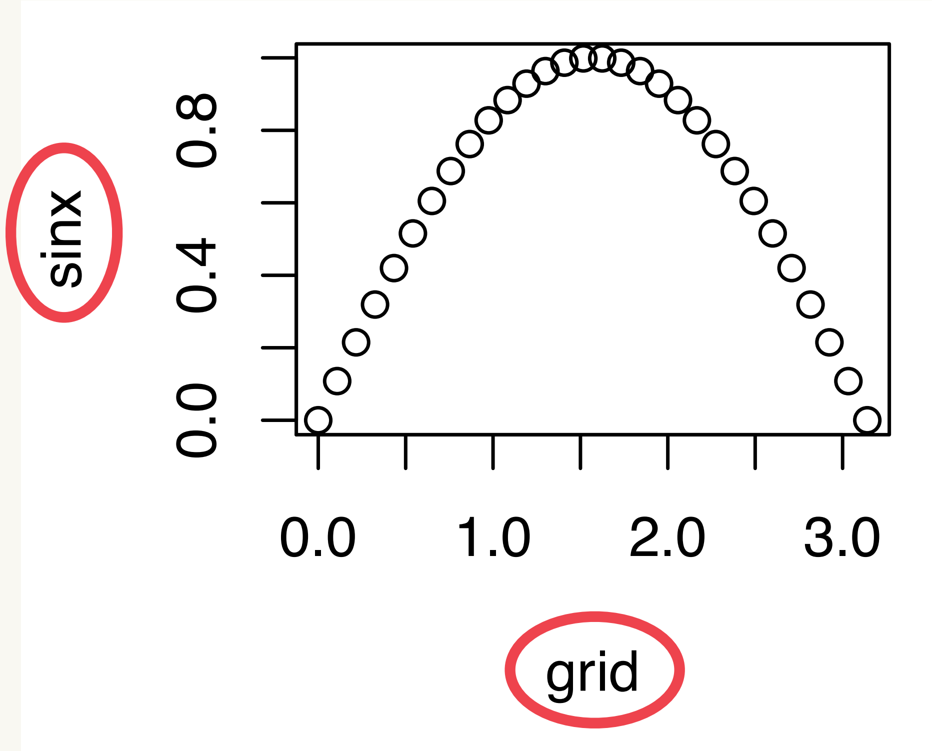
# Another motivation

# NSE commonly used for labelling

```r
grid <- seq(0, pi, , 30)
sinx <- sin(grid)

plot(grid, sinx)

# Inside plot:
xlabel <- deparse(subsitute(xlab))
```



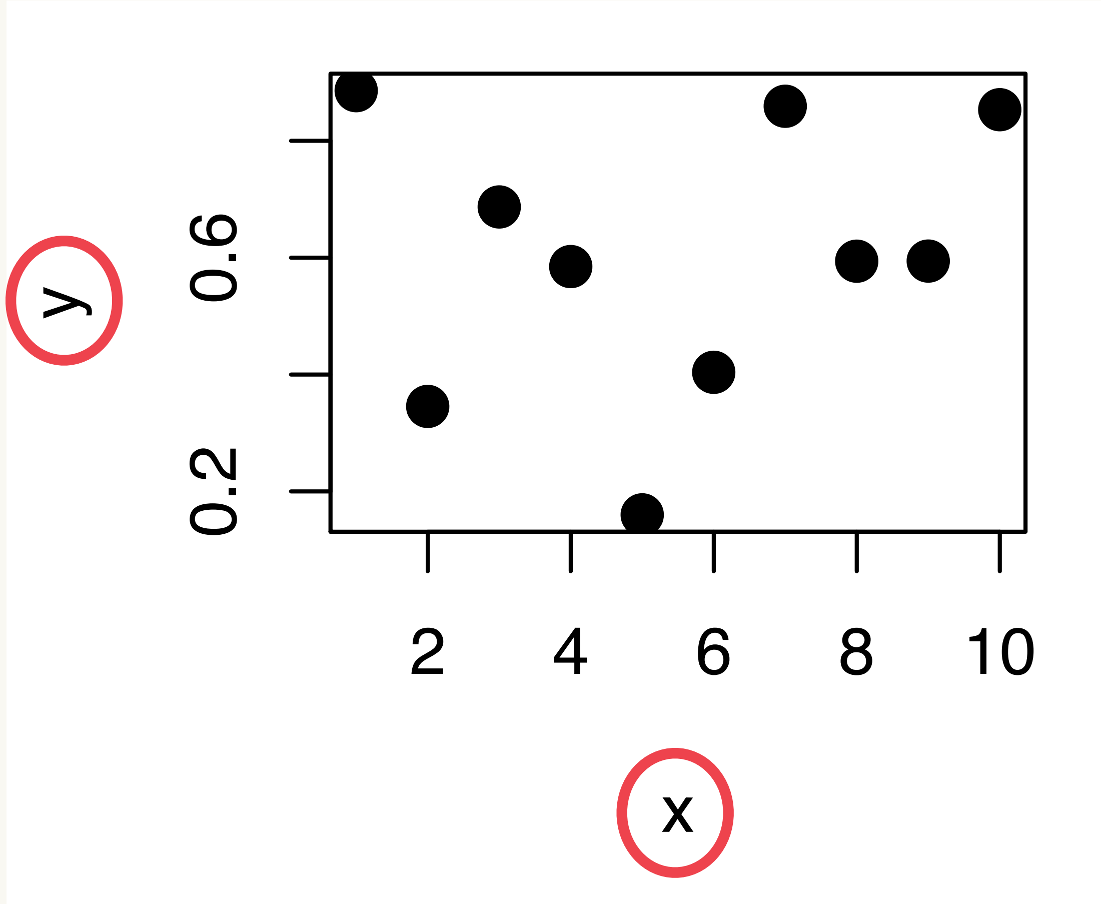
# Con: deparse() returns a vector!

```
deparse(quote({
  a + b
  c + d
}))

# Not a problem for plot, but I've been
# bitten by this many times in error messages
```

# Con: substitute() doesn't follow chain of promises

```
myplot <- function(x, y) {
  plot(x, y, pch = 20, cex = 2)
}
myplot(1:10, runif(10))
```



# lazyeval also provides some tools

```
# Like substitute, but finds "root" promise
expr_find(x)
expr_env(x, default_env)

# Couple of helpers to convert to strings
expr_text(x)
expr_label(x)
```

# Implementation is relatively straightforward

```
SEXP base_promise(SEXP promise, SEXP env) {
  while(TYPEOF(promise) == PROMSXP) {
    env = PRENV(promise);
    promise = PREXPR(promise);

    if (env == R_NilValue)
      break;

    if (TYPEOF(promise) == SYMSXP) {
      SEXP obj = Rf_findVar(promise, env);

      if (TYPEOF(obj) != PROMSXP)
        break;

      if (is_lazy_load(obj))
        break;

      promise = obj;
    }
  }

  return promise;
}
```

# Conclusion

1. Where possible, use **formulas** instead of NSE.
2. Provide **pronouns** to disambiguate.
3. Use **quasiquotation** to parameterise.

# lazyeval

https://github.com/hadley/lazyeval/

http://rpubs.com/hadley/lazyeval