Back at 1:30pm

Tidy evaluation:

Programming with ggplot2 and dplyr

May 2018

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

Writing functions

Rule of three: make a function if you've copy-pasted threes times

```
(df$a - min(df$a)) / (max(df$a) - min(df$a))
(df$b - min(df$b)) / (max(df$b) - min(df$b))
(df$c - min(df$c)) / (max(df$c) - min(df$c))
(df$d - min(df$d)) / (max(df$d) - min(df$c))
```

Rule of three: make a function if you've copy-pasted threes times

```
(df$a - min(df$a)) / (max(df$a) - min(df$a))
(df$b - min(df$b)) / (max(df$b) - min(df$b))
(df$c - min(df$c)) / (max(df$c) - min(df$c))
(df$d - min(df$d)) / (max(df$d) - min(df$c))
```

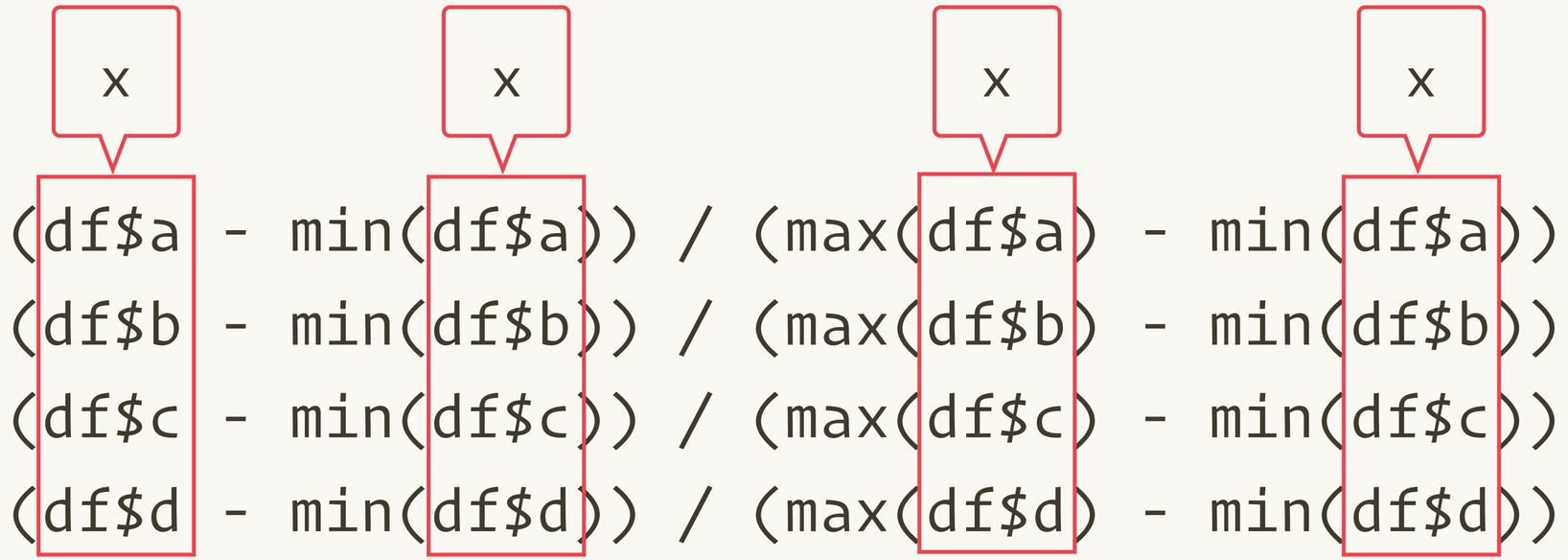
Rule of three: make a function if you've copy-pasted threes times

```
(df$a - min(df$a)) / (max(df$a) - min(df$a))
(df$b - min(df$b)) / (max(df$b) - min(df$b))
(df$c - min(df$c)) / (max(df$c) - min(df$c))
(df$d - min(df$d)) / (max(df$d) - min(df$d))
```

First, identify the parts that might change

```
(df$a - min(df$a)) / (max(df$a) - min(df$a))
(df$b - min(df$b)) / (max(df$b) - min(df$b))
(df$c - min(df$c)) / (max(df$c) - min(df$c))
(df$d - min(df$d)) / (max(df$d) - min(df$d))
```

Then give them names



Make the function template

```
rescale01 <- function(x) {

}
```

Then copy in one example

```
rescale01 <- function(x) {
  (df$a - min(df$a)) / (max(df$a) - min(df$a))
}
```

And use the variable

```
rescale01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}
```

And maybe refactor a little

```
rescale01 <- function(x) {
  rng <- range(x)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

And handle more cases

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1]))
}
```

Rule of three: make a function if you've copy-pasted threes times

```
(df$a - min(df$a)) / (max(df$a) - min(df$a))
(df$b - min(df$b)) / (max(df$b) - min(df$b))
(df$c - min(df$c)) / (max(df$c) - min(df$c))
(df$d - min(df$d)) / (max(df$d) - min(df$d))
```

Rule of three: make a function if you've copy-pasted threes times

```
rescale01(df$a)
rescale01(df$b)
rescale01(df$c)
rescale01(df$d)
```

Why create a function? Because a function:

1. Prevents inconsistencies
2. Emphasises what varies
3. Makes change easier
4. Can have informative name

Motivation

Let’s try with some dplyr code

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Your turn

Identify the parts that change.

Give them names.

Make a function.

Does it work?

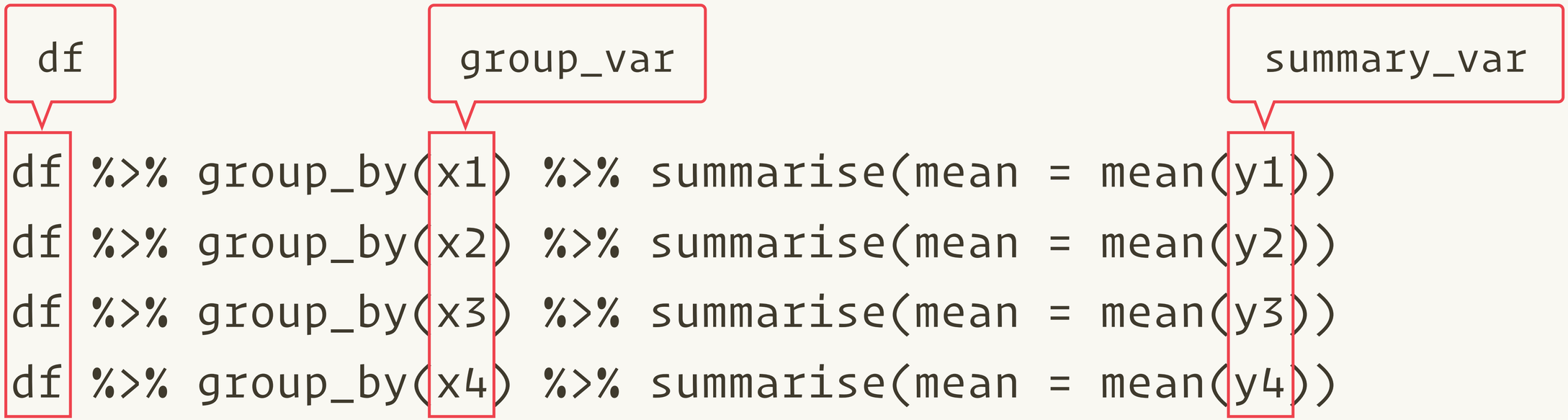
Let’s try with some dplyr code

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

First identify the parts that change

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Then give them names



Now make a function

```
grouped_mean <- function(df, group_var, summary_var) {
  df %>%
     group_by(group_var) %>%
     summarise(mean = mean(summary_var))
}
```

It doesn’t work 😭

```
grouped_mean <- function(df, group_var, summary_var) {
  df %>%
     group_by(group_var) %>%
     summarise(mean = mean(summary_var))
}

grouped_mean(mtcars, cyl, mpg)
#> Error: Column `group_var` is unknown
```

Vocabulary

We need some new vocabulary

Evaluated using usual R rules

```
(x - min(x)) / (max(x) - min(x))
```

```
mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))
```

Automatically **quoted** and evaluated in a "non-standard" way

You’re already familiar with this idea

```
df <- data.frame(
  y = 1,
  var = 2
)

df$y



var <- "y"
df$var
```

Predict the output!

$ automatically quotes the variable name

```
df <- data.frame(
  y = 1,
  var = 2
)

df$y
#> [1] 1

var <- "y"
df$var
#> [1] 2
```

If you want refer indirectly, must use [[instead

```
df <- data.frame(
  y = 1,
  var = 2
)

var <- "y"
df[[var]]
#> [1] 1
```

	Quoted	Evaluated
Direct	`df$y`	`???`
Indirect	`???`	`var <- "y"` `df[[var]]`

	Quoted	Evaluated
Direct	`df$y`	`df[["y"]]`
Indirect	`???`	`var <- "y"` `df[[var]]`

	Quoted	Evaluated
Direct	`df$y`	`df[["y"]]`
Indirect		`var <- "y"` `df[[var]]`

Identify which arguments are auto-quoted

```
library(MASS)

mtcars2 <- subset(mtcars, cyl == 4)

with(mtcars2, sum(vs))
sum(mtcars2$am)

rm(mtcars2)
```

Can’t tell? Try running the code

```
library(MASS)
#> Works

MASS
#> Error: object 'MASS' not found

# -> The 1st argument of library() is quoted
```

Can’t tell? Try running the code

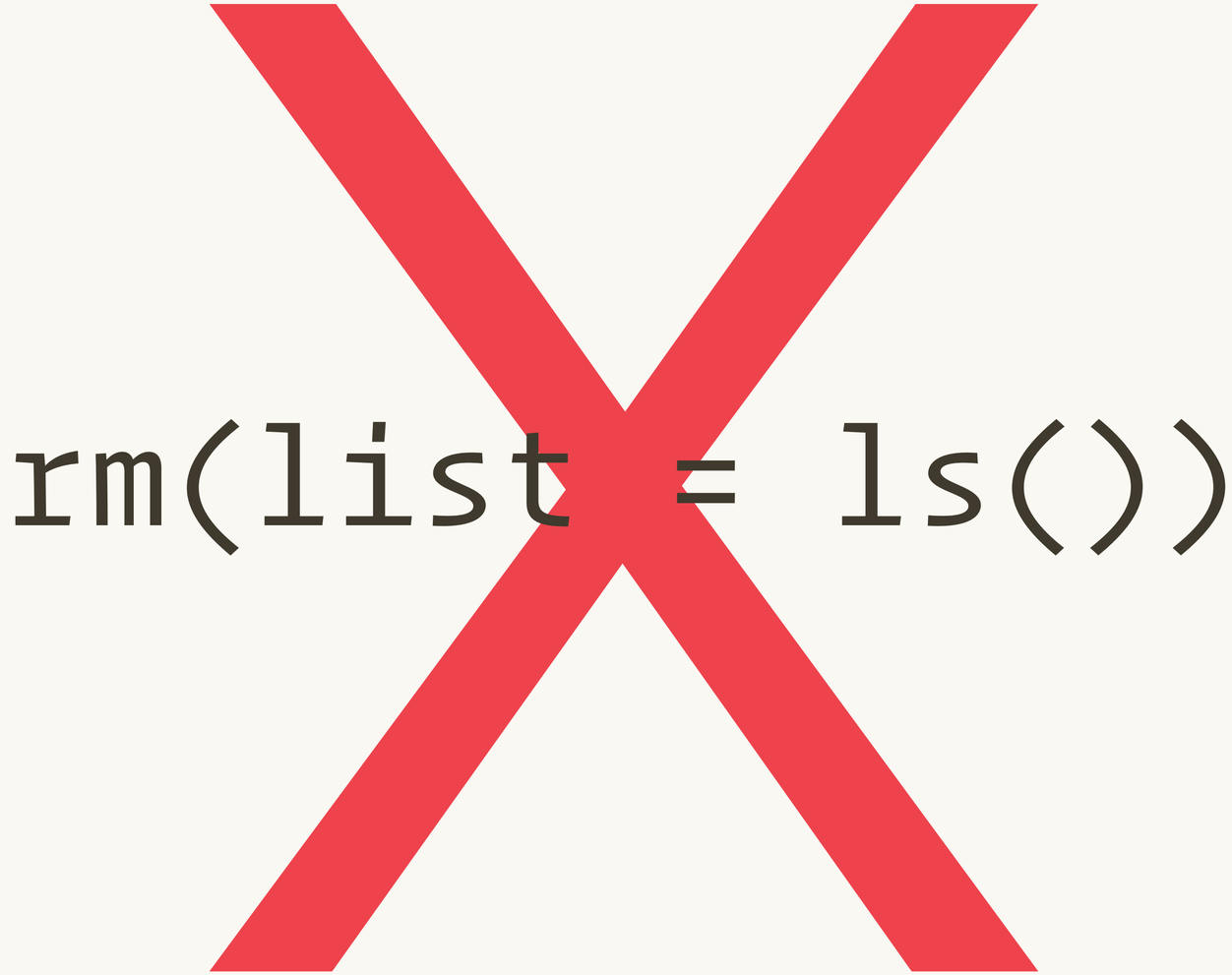
```
subset(mtcars, cyl == 4)
#> Works

cyl == 4
#> Error: object 'cyl' not found

# -> The 2nd argument of subset() is quoted
```

You can now identify the quoted arguments

```
library(MASS)

mtcars2 <- subset(mtcars, cyl == 4)

with(mtcars2, sum(vs))
sum(mtcars2$am)

rm(mtcars2)
```

Base R has 3 primary ways to “unquote”

Quoted/Direct	Evaluated/Indirect
`df$y`	`x <- "y"` `df[[x]]`
`library(MASS)`	`x <- "MASS"` `library(x, character.only = TRUE)`
`rm(mtcars)`	`x <- "mtcars"` `rm(list = x)`



https://www.tidyverse.org/articles/2017/12/workflow-vs-script/

Identify which arguments are auto-quoted

```
library(tidyverse)


mtcars %>% pull(am)


by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))


ggplot(by_cyl, aes(cyl, mpg)) +
  geom_point()
```

Identify which arguments are auto-quoted

```
library(tidyverse)

mtcars %>% pull(am)

by_cyl <- mtcars %>%
  group_by(cyl) %>%
  summarise(mean = mean(mpg))

ggplot(by_cyl, aes(cyl, mpg)) +
  geom_point()
```

	Quoted	Evaluated	Tidy
Direct	`df$y`	`df[["y"]]`	`pull(df, y)`
Indirect	😢	`var <- "y"` `df[[var]]`	`???`

	Quoted	Evaluated	Tidy
Direct	`df$y`	`df[["y"]]`	`pull(df, y)`
Indirect	😢	`var <- "y"` `df[[var]]`	`var <- quo(y)` `pull(df, !!var)`

Everywhere in the tidyverse uses !! to unquote

Pronounced bang-bang

```
x_var <- quo(cyl)
y_var <- quo(mpg)

by_cyl <- mtcars %>%
  group_by(!!x_var) %>%
  summarise(mean = mean(!!y_var))

ggplot(by_cyl, aes(!!x_var, !!y_var)) +
  geom_point()
```

Wrapping quoting functions

New: Identify quoted vs. evaluated arguments

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

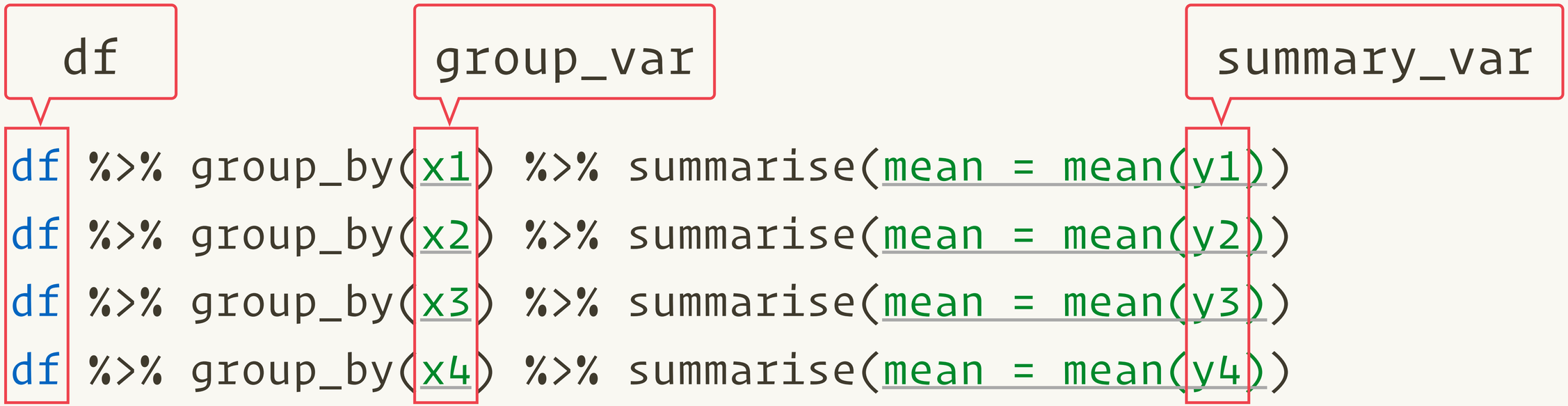
New: Identify quoted vs. evaluated arguments

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

Then identify the parts that could change

```
df %>% group_by(x1) %>% summarise(mean = mean(y1))
df %>% group_by(x2) %>% summarise(mean = mean(y2))
df %>% group_by(x3) %>% summarise(mean = mean(y3))
df %>% group_by(x4) %>% summarise(mean = mean(y4))
```

These become the function arguments



Next write the function template & identify quoted arguments

```
grouped_mean <- function(df, group_var, summary_var) {



   df %>%
     group_by(group_var) %>%
     summarise(mean = mean(summary_var))
}
```

New: Wrap every quoted argument in enquo()

```
grouped_mean <- function(df, group_var, summary_var) {
   group_var <- enquo(group_var)
   summary_var <- enquo(summary_var)

   df %>%
     group_by(group_var) %>%
     summarise(mean = mean(summary_var))
}
```

New: And then unquote with !!

```
grouped_mean <- function(df, group_var, summary_var) {
   group_var <- enquo(group_var)
   summary_var <- enquo(summary_var)

   df %>%
     group_by(!!group_var) %>%
     summarise(mean = mean(!!summary_var))
}
```

```r
grouped_mean(mtcars, cyl, mpg)

grouped_mean <- function(df, group_var, summary_var) {
   group_var <- enquo(group_var)
   summary_var <- enquo(summary_var)

   df %>%
     group_by(!!group_var) %>%
     summarise(mean = mean(!!summary_var))
}
```

```r
grouped_mean(mtcars, cyl, mpg)

grouped_mean <- function(df, group_var, summary_var) {
   group_var <- quo(cyl)
   summary_var <- quo(mpg)

   df %>%
     group_by(!!group_var) %>%
     summarise(mean = mean(!!summary_var))
}
```

```r
grouped_mean(mtcars, cyl, mpg)

grouped_mean <- function(df, group_var, summary_var) {
   group_var <- quo(cyl)
   summary_var <- quo(mpg)

   df %>%
     group_by(cyl) %>%
     summarise(mean = mean(mpg))
}
```

Is it worth it?

It saves a lot of typing

```
filter(diamonds, x > 0 & y > 0 & z > 0)


# vs
diamonds[
  diamonds$x > 0 &
  diamonds$y > 0 &
  diamonds$z > 0,
]
```

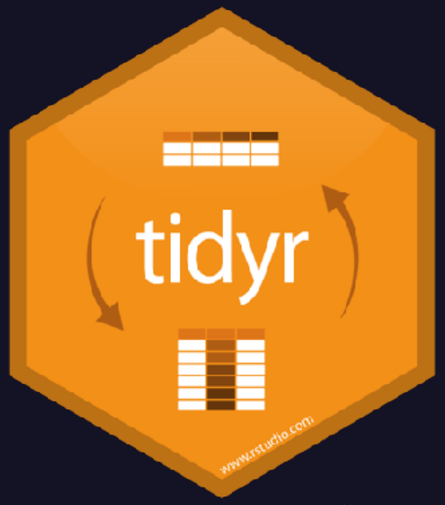
It saves a lot of typing

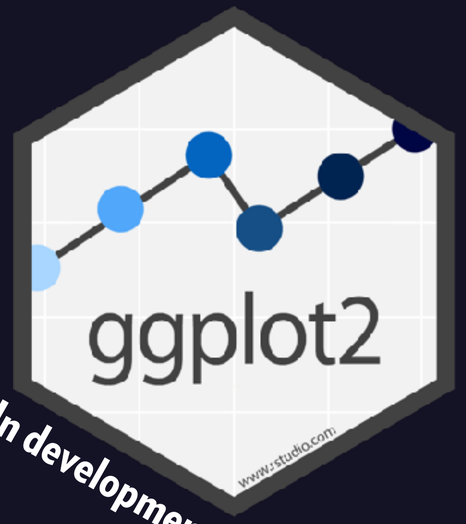
```
filter(diamonds, x > 0 & y > 0 & z > 0)


# vs
diamonds[
  diamonds["x"] > 0 &
  diamonds["y"] > 0 &
  diamonds["z"] > 0,
]
```

And makes it possible to translate to other languages

```
mtcars_db %>%
  filter(cyl > 2) %>%
  select(mpg:hp) %>%
  head(10) %>%
  show_query()

#> SELECT `mpg`, `cyl`, `disp`, `hp`
#> FROM `mtcars`
#> WHERE (`cyl` > 2.0)
#> LIMIT 10
```

Tidy evaluation = principled NSE





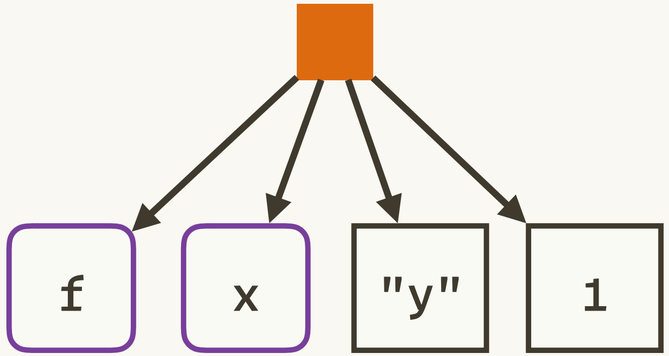
In development

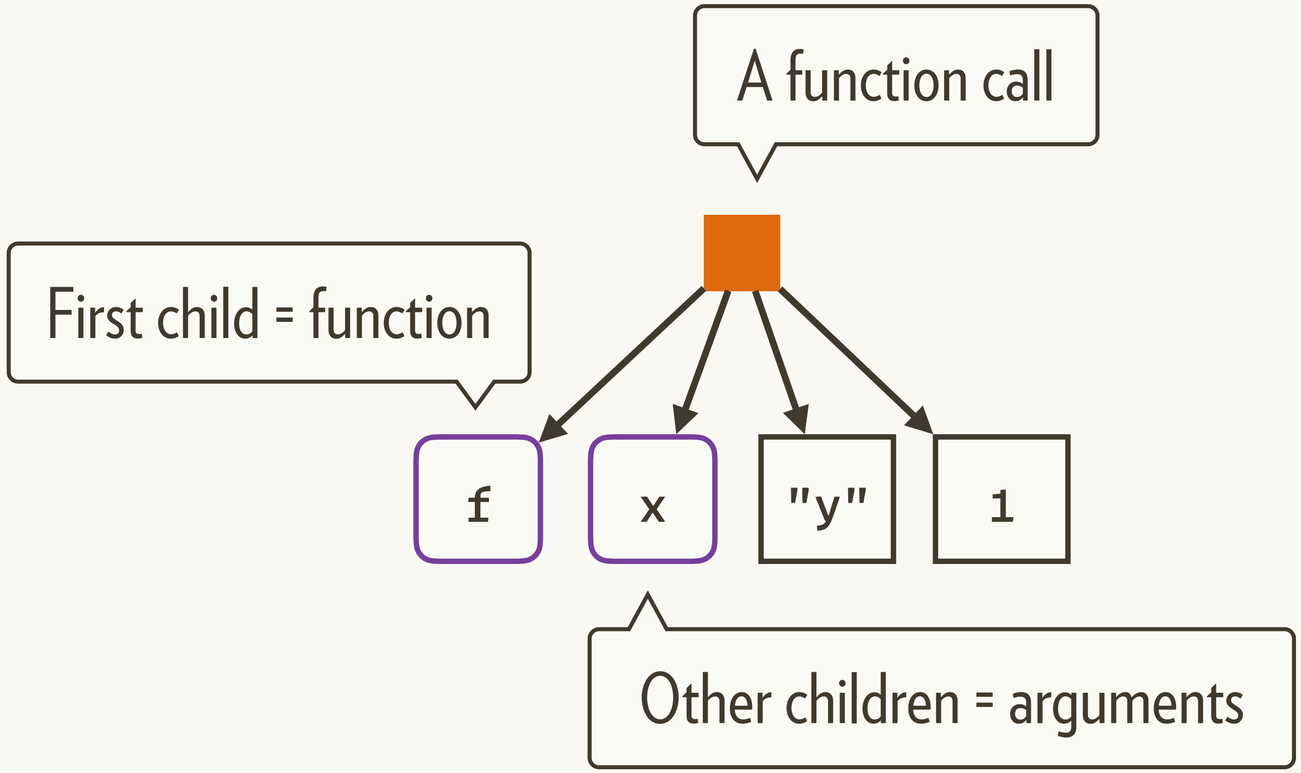
Now for some ~~game~~ theory

1. R code is a tree
2. Unquoting builds trees
3. Environments map names to values

R code is a tree

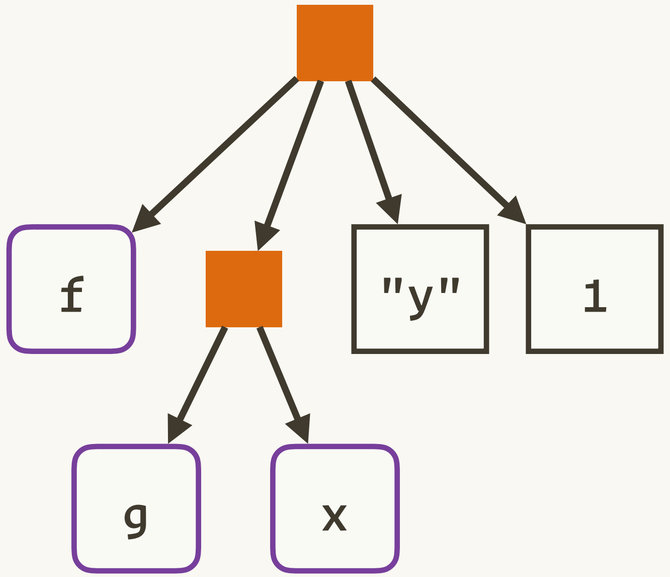
```
f(x, "y", 1)
```




A function call
First child = function
f
x
"y"
1
Other children = arguments

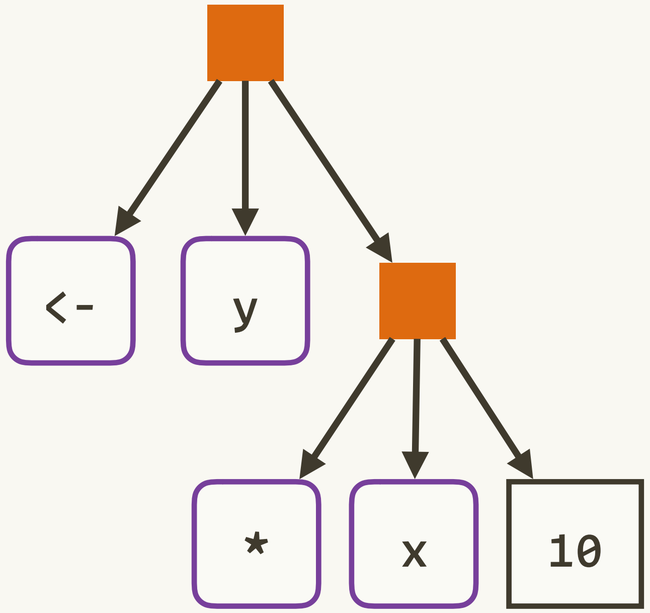
More complex calls have multiple levels

```
f(g(x), "y", 1)
```



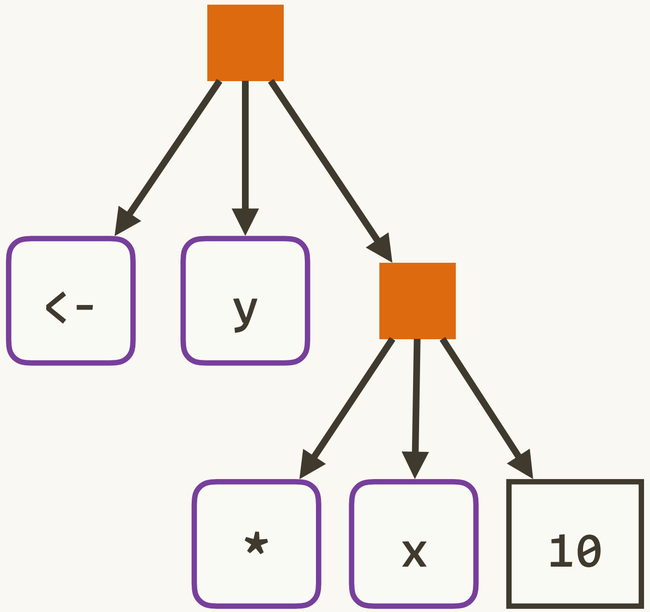
Every expression has a tree

```
y <- x * 10
```



Because every expression can be rewritten

```
`<-`(y, `*`(x, 10))
```



You can see this yourself with lobstr::ast()

```
> lobstr::ast(if(x > 5) y + 1)
■─`if`
├─■─`>`
│ ├─x
│ └─5
└─■─`+`
  ├─y
  └─1
```

Your turn

```
library(lobstr)

# Compare to my hand drawn diagrams
ast(f(x, "y", 1))
ast(y <- x * 10)

# What does this tree tell you?
ast(function(x, y) {
  if (x > y) {
    x
  } else {
    y
  }
})
```

What isn’t in the AST?

```
ast(1          + 2)
ast({
  1
  # comment
  2
})
```

Unquoting builds trees

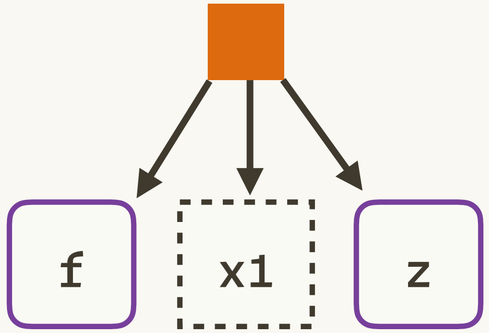
expr() captures your expression

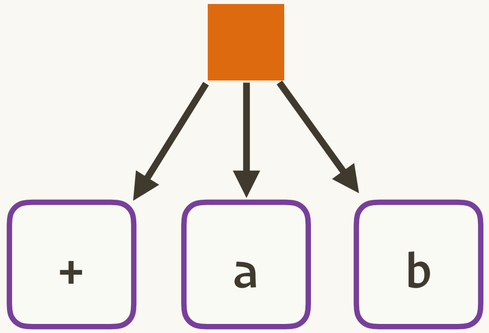
```
library(rlang)

expr(y + 1)
#> y + 1
```

Unquoting allows you to build your own trees

```
x1 <- expr(a + b)
expr(f(!!x1, z))
#> f(a + b, z)

# !! is called the unquoting operator
# And is pronounced bang-bang
```

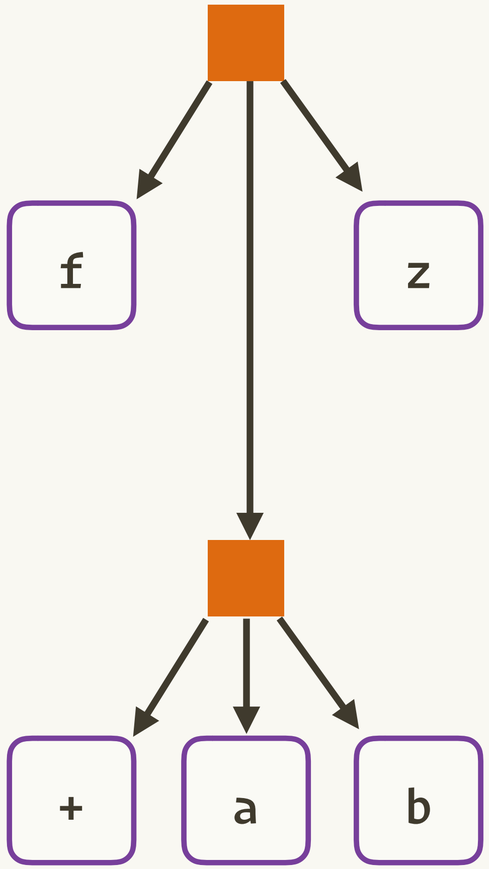
```
expr(f(!!x1, z))
```



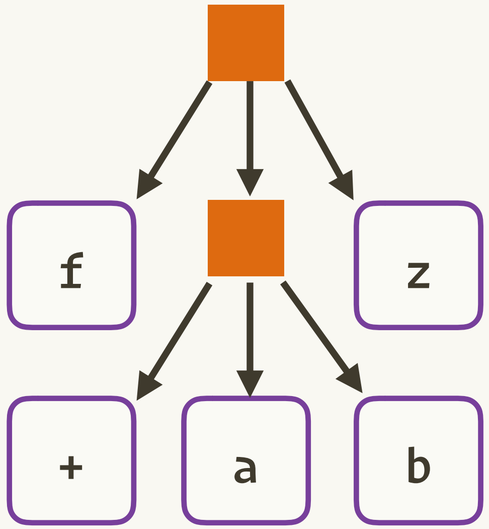



```
x1 <- expr(a + b)
```

```
expr(f(!!x1, z))
```



```
expr(f(!!x1, z))
```



Predict what this code will return

```
ex1 <- expr(x + y)

ex2 <- expr(!!ex1 + z)

ex3 <- expr(1 / !!ex1)
```

Predict what this code will return

```
ex1 <- expr(x + y)
# x + y
ex2 <- expr(!!ex1 + z)


ex3 <- expr(1 / !!ex1)
```

Predict what this code will return

```
ex1 <- expr(x + y)
# x + y
ex2 <- expr(!!ex1 + z)
# x + y + z
ex3 <- expr(1 / !!ex1)
```

Predict what this code will return

```
ex1 <- expr(x + y)
# x + y
ex2 <- expr(!!ex1 + z)
# x + y + z
ex3 <- expr(1 / !!ex1)
# 1 / (x + y)
# Not 1 / x + y
```

```
ex2 <- expr(ex1 + z)
ex1 + z
```

Recreate these expressions

```
# These are terrible
(x + y) / (y + z)
-(x + z) ^ (y + z)
(x + y) + (y + z) - (x + y)
atan2(y = x + y, x = y + z)
foo(x + y, y + z)

# using
xy <- expr(x + y)
xz <- expr(x + z)
yz <- expr(y + z)
f <- expr(foo)
```

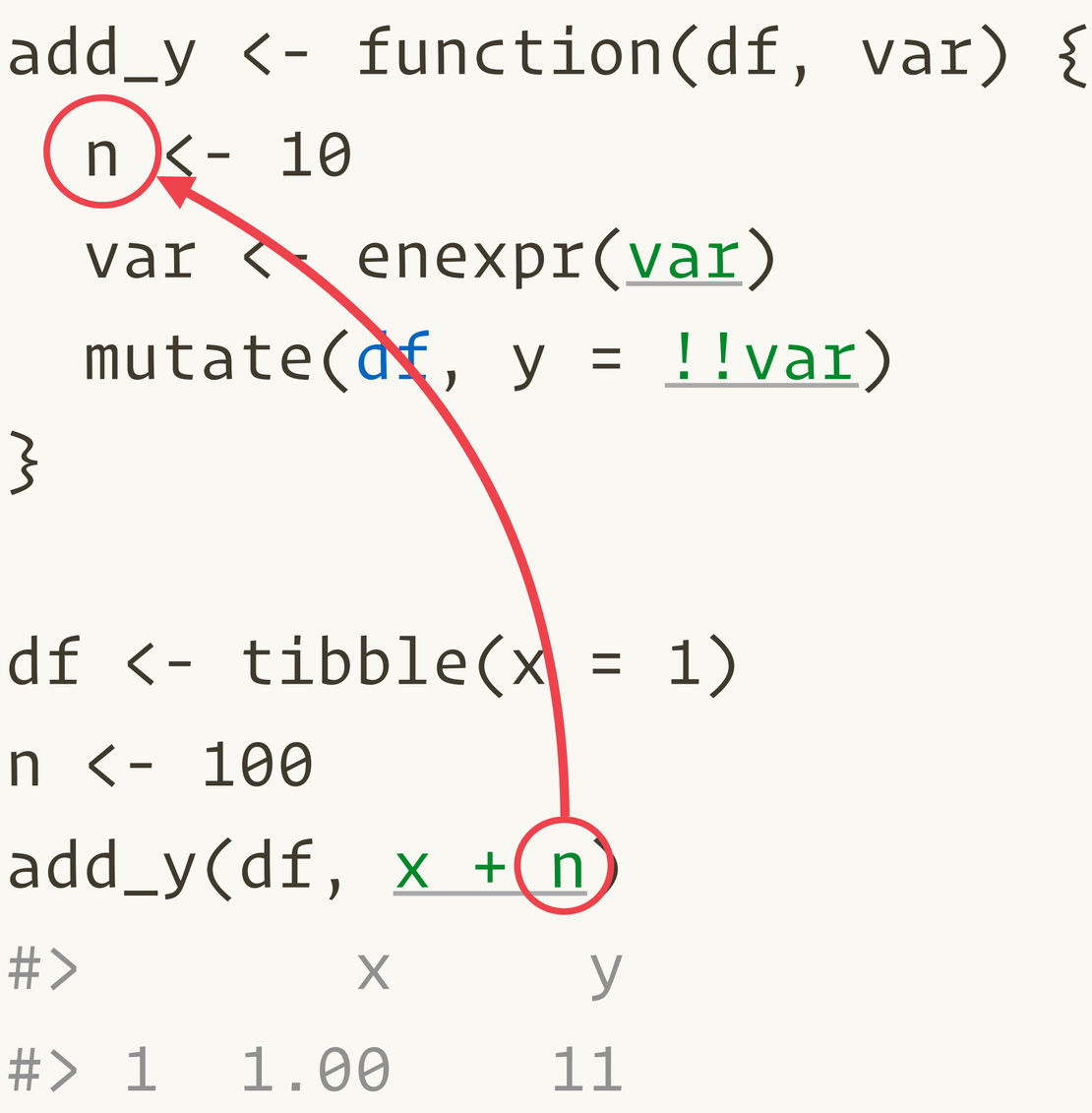
enexpr() lets you capture user expressions

```
# expr() quotes your expression
f1 <- function(z) expr(z)
f1(a + b)
#> z


# enexpr() quotes user's expression
f2 <- function(z) enexpr(z)
f2(x + y)
#> x + y
```

Environments map names to values

Capturing just expression isn't enough

```
add_y <- function(df, var) {
  n <- 10
  var <- enexpr(var)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>       x     y
#> 1  1.00    11
```



```
add_y <- function(df, var) {
  n <- 10
  var <- enexpr(var)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>      x     y
#> 1 1.00    11
```

Capturing just expression isn't enough

```
add_y <- function(df, var) {
  n <- 10
  var <- enexpr(var)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>       x     y
#> 1  1.00    11
```

```r
add_y <- function(df, var) {
  n <- 10
  var <- expr(x + n)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>       x     y
#> 1  1.00    11
```

```
add_y <- function(df, var) {
  n <- 10
  var <- expr(x + n)
  mutate(df, y = x + n)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>       x     y
#> 1  1.00    11
```

quo() captures expression and environment

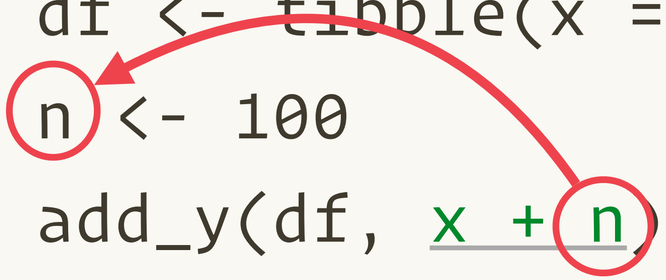
```
# quo() quotes your expression
f1 <- function(z) quo(z)
f1(a + b)
#> <quosure>
#>  expr: ^z
#>  env:  0x10d3b9308

# enquo() quotes user's expression
f2 <- function(z) enquo(z)
f2(x + y)
#> <quosure>
#>  expr: ^x + y
#>  env:  0x10d3b9309
```

	Function author	Function user
Expression	`expr(x)`	`enenxpr(x)`
Expression + environment	`quo(x)`	`enquo(x)`

Think enrich

```r
add_y <- function(df, var) {
  n <- 10
  var <- enquo(var)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>       x     y
#> 1  1.00   101
```

```
add_y <- function(df, var) {
  n <- 10
  var <- enquo(var)
  mutate(df, y = !!var)
}

df <- tibble(x = 1)
n <- 100
add_y(df, x + n)
#>         x      y
#> 1    1.00    101
```



Key pattern is to quote and unquote

```
df <- data.frame(x = 1:5, y = 5:1)
filter(df, abs(x) > 1e-3)
filter(df, abs(y) > 1e-3)
filter(df, abs(z) > 1e-3)

my_filter <- function(df, var) {
  var <- enquo(var)
  filter(df, abs(!!var) > 1e-3)
}
my_filter(df, x)
```

Quote

Unquote

Reduce the duplication here

```
df <- data.frame(
  g = rep(c("a", "b", "c"), c(3, 2, 2)),
  b = runif(7),
  a = runif(7),
  c = runif(7)
)

summarise(df, mean = mean(a), sd = sd(a), n = n())
summarise(df, mean = mean(b), sd = sd(b), n = n())
summarise(df, mean = mean(c), sd = sd(c), n = n())
```

```r
stat_sum <- function(df, var) {
  var <- enquo(var)

  summarise(df,
    mean = mean(!!var),
    sd = sd(!!var),
    n = n()
  )
}
```

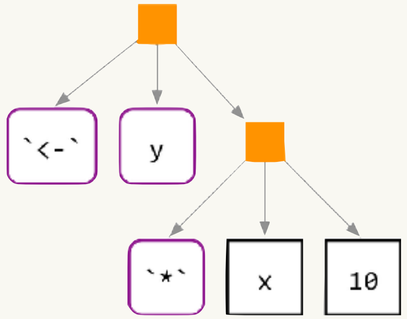
```
stat_sum <- function(df, var, summary_funs =
funs(mean, sd)) {
  var <- enquo(var)

  summarise_at(df, vars(!!var), summary_funs)
}
```

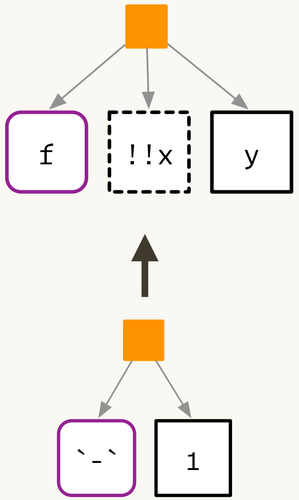
Learning more

Theory

https://youtu.be/nERXS3ssntw

Code is a tree

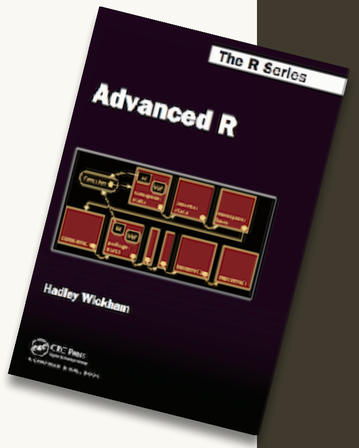




`enquo()`

Build trees with unquoting

https://adv-r.hadley.nz/expressions.html
https://adv-r.hadley.nz/quasiquotation.html
https://adv-r.hadley.nz/evaluation.html



Practice