

Tidy evaluation:

Programming with ggplot2 and dplyr

April 2019

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))
```

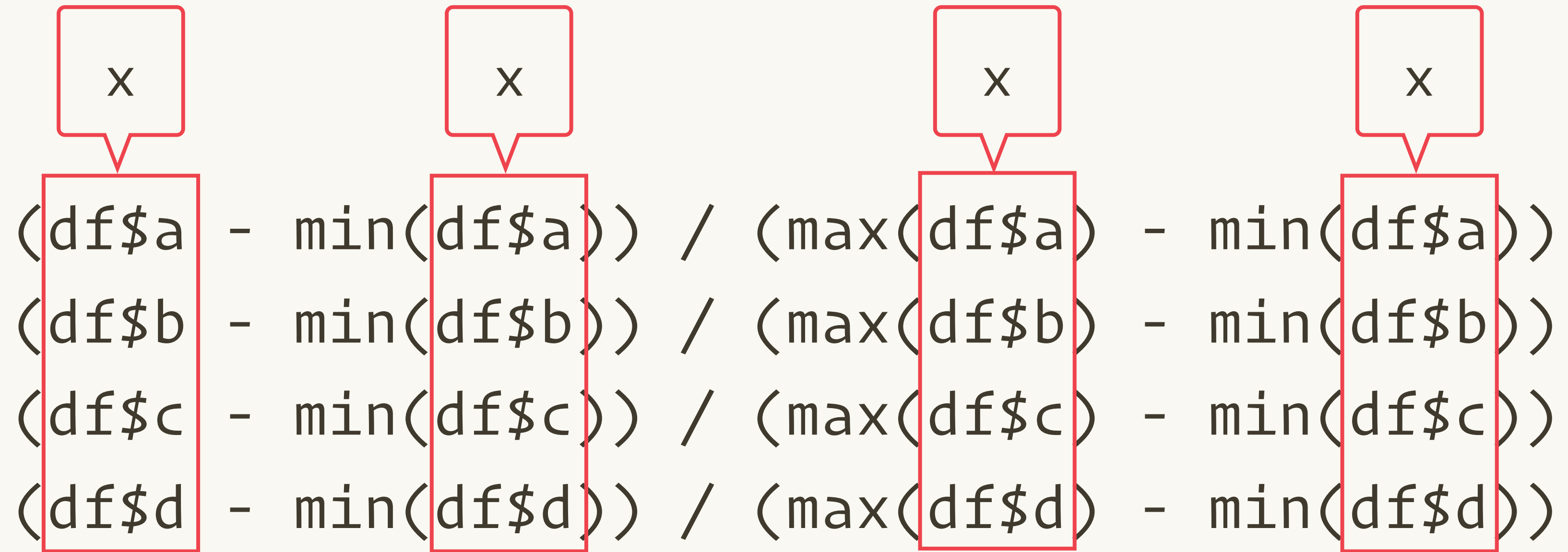
```
(df$c - min(df$c)) / (max(df$c) - min(df$c))
```

```
(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



The diagram shows four identical mathematical expressions arranged horizontally. Each expression is enclosed in a red rectangular box. Above each box is a red speech bubble containing the letter 'x'. The expressions are as follows:

$$\begin{aligned} & \left(\text{df\$a} - \min(\text{df\$a}) \right) / \left(\max(\text{df\$a}) - \min(\text{df\$a}) \right) \\ & \left(\text{df\$b} - \min(\text{df\$b}) \right) / \left(\max(\text{df\$b}) - \min(\text{df\$b}) \right) \\ & \left(\text{df\$c} - \min(\text{df\$c}) \right) / \left(\max(\text{df\$c}) - \min(\text{df\$c}) \right) \\ & \left(\text{df\$d} - \min(\text{df\$d}) \right) / \left(\max(\text{df\$d}) - \min(\text{df\$d}) \right) \end{aligned}$$

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.

Why doesn't 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

df

group_var

summary_var

```
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))
```

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	<code>df\$<u>y</u></code>	<code>???</code>
Indirect	<code>???</code>	<code>var <- "y"</code> <code>df[[<u>var</u>]]</code>

	Quoted	Evaluated
Direct	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>
Indirect	<code>???</code>	<code>var <- "y"</code> <code>df[<u>var</u>]</code>

	Quoted	Evaluated
Direct	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>
Indirect		<code>var <- "y"</code> <code>df[[var]]</code>

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
<code>df\$<u>y</u></code>	<pre>x <- "y" df[[x]]</pre>
<code>library(<u>MASS</u>)</code>	<pre>x <- "MASS" library(x, character.only = TRUE)</pre>
<code>rm(<u>mtcars</u>)</code>	<pre>x <- "mtcars" rm(list = x)</pre>



```
rm(list = ls())
```

<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	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>	<code>pull(df, <u>y</u>)</code>
Indirect		<code>var <- "y"</code> <code>df[[<u>var</u>]]</code>	<code>???</code>

	Quoted	Evaluated	Tidy
Direct	<code>df\$<u>y</u></code>	<code>df[["y"]]</code>	<code>pull(df, <u>y</u>)</code>
Indirect		<code>var <- "y"</code> <code>df[[<u>var</u>]]</code>	<code>var <- quo(<u>y</u>)</code> <code>pull(df, !!<u>var</u>)</code>

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

df

group_var

summary_var

```
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))
```

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))  
}
```

Use the expression stored inside the variable, not literally "summary_var"

```
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))  
}
```

```
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))  
}
```

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

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

```
  df %>%
```

```
    group_by(cyl) %>%
```

```
    summarise(mean = mean(mpg))
```

```
}
```

Coming soon...

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

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

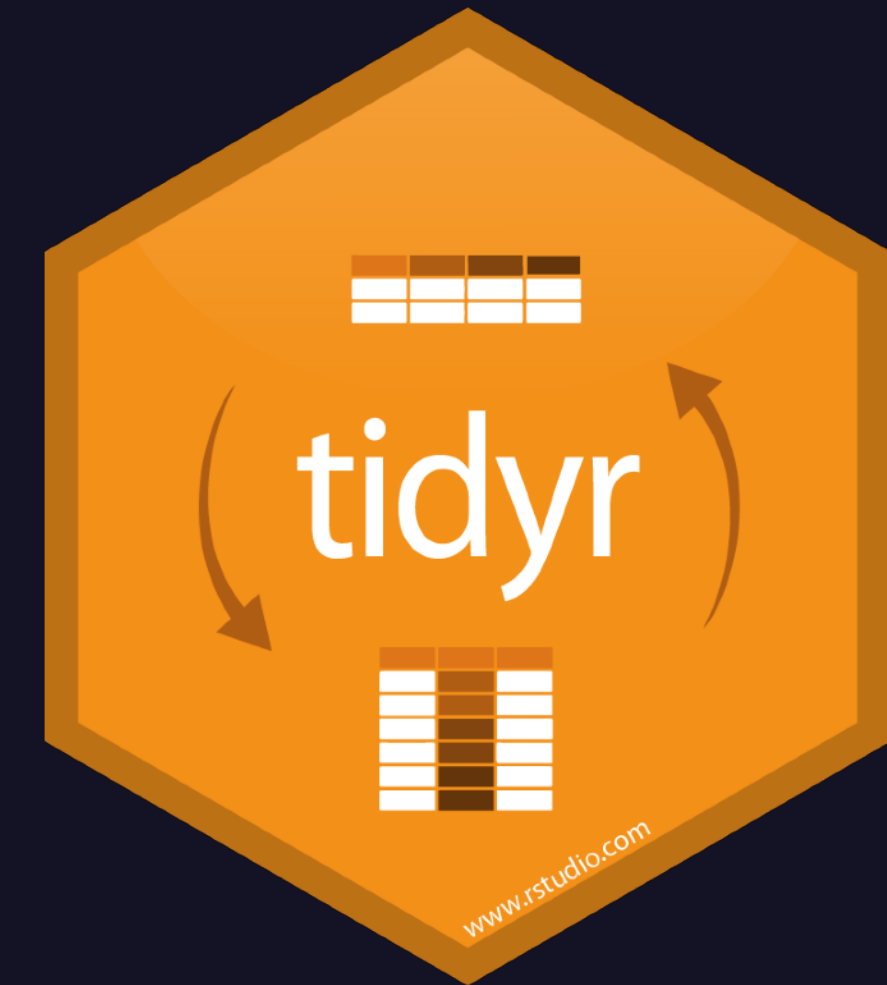
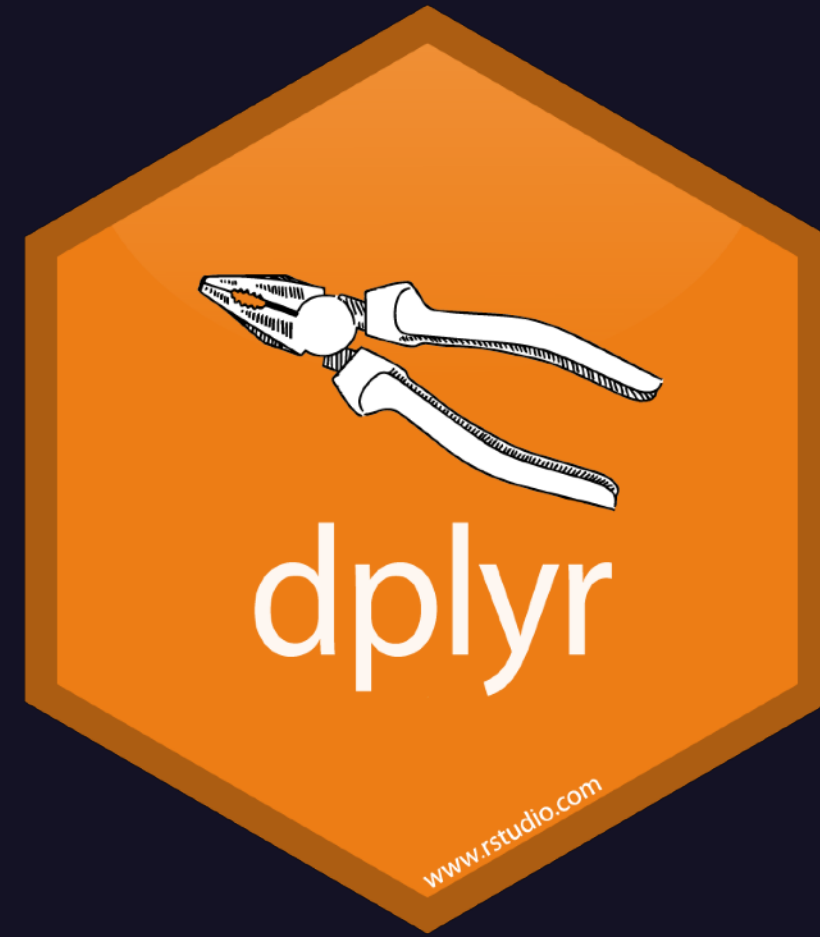
```
diamonds[  
  diamonds[["x"]] > 0 &  
  diamonds[["y"]] > 0 &  
  diamonds[["z"]] > 0,  
]
```

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



The underlying theory is elegant (IMO)

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

Practice

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())
```

```
stat_sum <- function(df, var) {  
  var <- enquos(var)  
  
  summarise(df,  
    mean = mean(!var),  
    sd = sd(!var),  
    n = n()  
  )  
}
```

Your turn

```
# It's often useful to compute a proportion
# of a grouped sum. Complete the function below
# to simplify this useful pattern

mtcars %>% count(cyl) %>% mutate(prop = n / sum(n))

prop <- function(df, x = n) {
  x <- enquo(x)
  ...
}
```



```
prop <- function(df, x = n) {  
  x <- enquos(x)  
  df %>% mutate(prop = !!x / sum(!!x, na.rm = TRUE))  
}
```

```
prop <- function(df, x = n) {  
  x <- enquos(x)  
  df %>% mutate(prop = prop.table(!!x))  
}
```

Make a reusable function for this pattern

```
counts <- starwars %>%  
  group_by(g = homeworld) %>%  
  summarise(n = n()) %>%  
  head(10) %>%  
  mutate(g = reorder(g, n))
```

```
counts %>%  
  ggplot(aes(g, n)) +  
  geom_col() +  
  coord_flip() +  
  xlab("homeworld")
```

See template on next slide

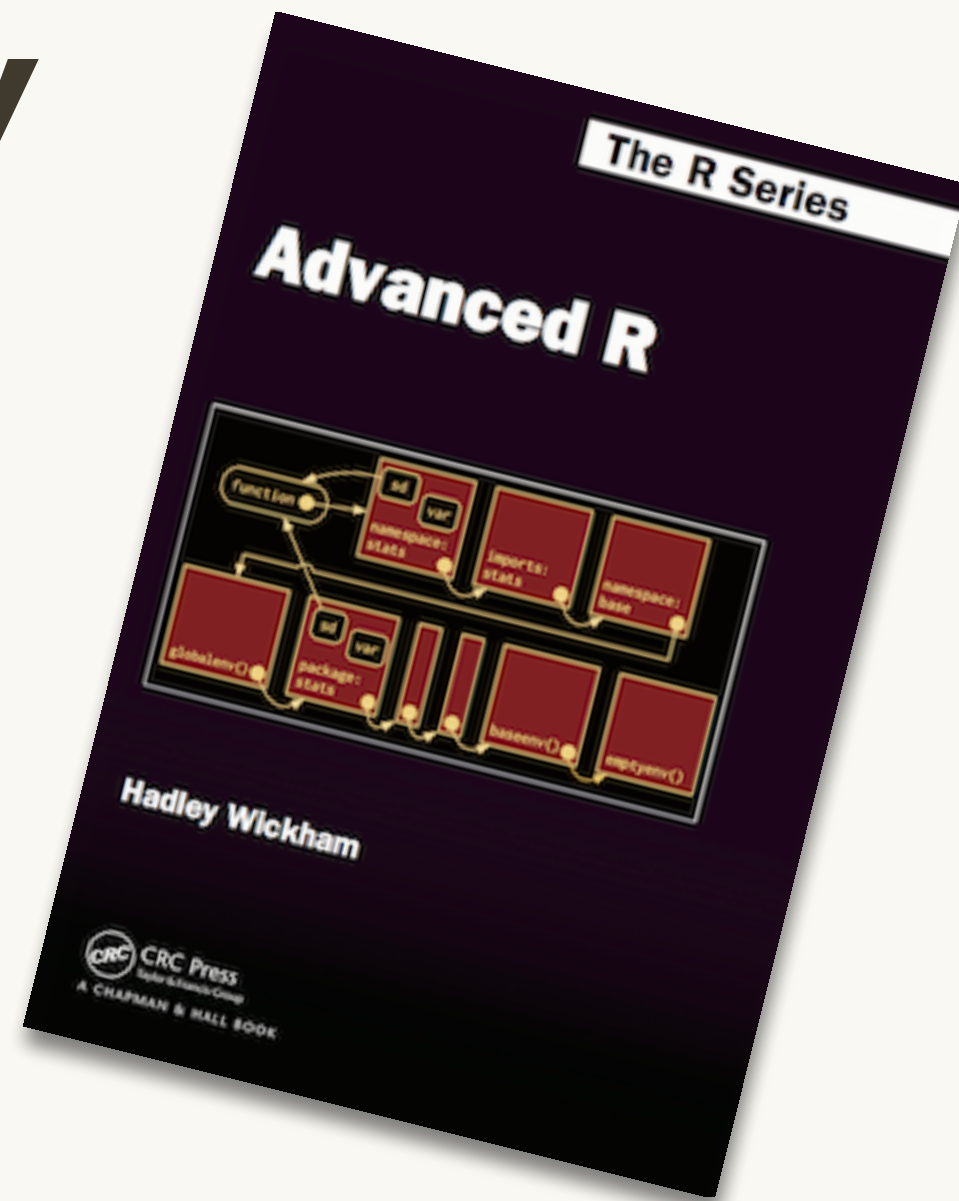
<https://twitter.com/JustTheSpring/status/1082515899821617152>

```
top_n <- function(df, x, n = 10) {  
  
}
```

```
# Challenge: can you change the basic approach  
# to better handle ties?
```

Learning more

Theory



<https://adv-r.hadley.nz/expressions.html>

<https://adv-r.hadley.nz/quasiquotation.html>

<https://adv-r.hadley.nz/evaluation.html>

<https://youtu.be/nERXS3ssntw>

Practice

<https://tidyeval.tidyverse.org>

(still a work in progress)

Tidy evaluation with rlang : : CHEAT SHEET

Vocabulary

Tidy Evaluation (Tidy Eval) is not a package, but a framework for doing non-standard evaluation (i.e. delayed evaluation) that makes it easier to program with tidyverse functions.

Symbol - a name that represents a value or object stored in R (i.e. `symbol(expr)`)

Environment - a list-like object that binds symbols (names) to objects stored in memory. Each env contains a link to a second, **parent** env, which creates a chain, or search path, of environments (i.e. `environment(current_env())`)

rlang::caller_env() Returns calling env of the function it is in.

rlang::child_env(parent, ...) Creates new env as child of parent. Also **env**.

rlang::current_env() Returns execution env of the function it is in.

Constant - a bare value (i.e. an atomic vector of length 1), e.g. `burn_0bms()`

Call object - a vector of symbols/constants/calls that begins with a function name, possibly followed by arguments, e.g. `collapse()`

Code - a sequence of symbols/constants/calls that will return a result if evaluated. Code can be:

1. Evaluated immediately (Standard Eval)
2. Quoted to use later (Non-standard Eval)

Expression - an object that stores quoted code without evaluating it, i.e. `expression(expr)`

Quosure - an object that stores both quoted code (without evaluating it) and the code's environment (i.e. `quosure(expr + b)`)

rlang::quo_get_env(quo) Return the environment of a quosure.

rlang::quo_set_env(quo, env) Set the environment of a quosure.

rlang::quo_get_expr(quo) Return the expression of a quosure.

Expression Vector - a list of pieces of quoted code created by base R's expression and parse functions. Not to be confused with **expression**.

Quoting Code

Quote code in one of two ways (if in doubt use a quosure):

QUOSURES

rlang::quos(expr) Quote contents as a quosure. Also **quos** to quote multiple expressions. `a <- 1, b <- 2, q <- quote + b, q <- quos(b)`

rlang::enquo(arg) Call from within a function to quote what the user passed to an argument as a quosure. Also **enquos** for multiple args. `quote_these <- function(...) enquos(...)`

rlang::new_quosure(expr, env = caller_env()) Build a quosure from a quoted expression and an environment. `new_quosure(expr + b, current_env())`

Parsing and Deparsing

Parse - Convert a string to a saved expression.

rlang::parse_expr() Convert a string to an expression. Also **parse_exprs**, **sym**, **parse_quos**. `parse_quos <- parse_exprs("a+b")`

rlang::expr_text(expr, width = 60L, min_col = 10L) Convert expr to a string. Also **quo_name**. `expr_text(a)`

Building Calls

rlang::call2(fn, ..., ns = NULL) Create a call from a function and a list of args. Use **exec** to create and then evaluate the call. (See back page for full args - index 4, base 1-2)

log `k = 4, base = 2`

2

`call2("log", x = 4, base = 2)`
`call2("log", 4)`
`exec("log", x = 4, base = 2)`
`exec("log", 4)`

Quoted Expression

An expression that has been saved by itself. A quoted expression can be evaluated later to return a result that will depend on the environment it is evaluated in.

rlang::expr(expr) Quote contents. Also **exprs** to quote multiple expressions. `a <- 1, b <- 2, e <- expr(a + b), e <- expr(b, a + b)`

rlang::enexpr(arg) Call from within a function to quote what the user passed to an argument as a symbol, accepts strings. Also **enexprs**. `quote_these <- function(...) enexprs(...)`

rlang::enexprs() Call from within a function to quote what the user passed to an argument as a symbol, accepts strings. Also **enexprs**. `quote_name <- function(name) enexprs(name)`
`quote_name <- function(...) enexprs(...)`

Evaluation

To evaluate an expression, R:

1. Looks up the symbols in the expression in the active environment (or a supplied env), followed by the environment's parents
2. Executes the calls in the expression

The result of an expression depends on which environment it is evaluated in.

QUOTED EXPRESSION

rlang::eval_bare(expr, env = parent.frame()) Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval_bare() stored environment, eval_bare() stored environment.

QUOSURES (and quoted exprs)

rlang::eval_tidy(expr, data = NULL, env = caller_env()) Evaluate expr in env, using data as a **data mask**. Will evaluate quosures in their stored environment, eval_bare() stored environment, eval_bare() stored environment.

Data Mask - If data is non-NULL, eval_tidy() inverts data into the search path before env, matching symbols to names in data. Use the pronoun **data** to force a symbol to be matched in data, and if (see back) to force a symbol to be matched in the environment.

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 864-449-1212 • rstudio.com • Learn more at <https://tidyeval.tidyverse.org> • rlang 4.0.0 • Updated: 2020-11

This work is licensed as
Creative Commons
Attribution-ShareAlike 4.0
International

To view a copy of this license, visit
<https://creativecommons.org/licenses/by-sa/4.0/>