

Functions

July 2018

Charlotte Wickham

@cvwickham

cwickham@gmail.com

cwick.co.nz

Adapted from *Tidy Tools* by Hadley Wickham



What does this code do?

```
df <- tibble::tibble(  
  a = c(rnorm(9), -99),  
  b = c(-999, -99, rnorm(8)),  
  c = c(0, rnorm(9)),  
  d = rnorm(10)  
)
```

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

Writing functions

The when and the how

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 illustrates four identical mathematical expressions arranged horizontally. Each expression is a ratio of two differences of minimum and maximum values. The first term of each ratio, `(df$a - min(df$a))`, is enclosed in a red rectangular box. Above each of these four boxes is a red speech bubble containing the letter 'x'. The four expressions are separated by a slash `/`, and the entire set is followed by a minus sign `-` and another identical set of four expressions. The second set also has red boxes around its first terms and red speech bubbles with 'x' above them.

x

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

x

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

x

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

x

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

Make the function template

function name

argument name

```
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 argument name

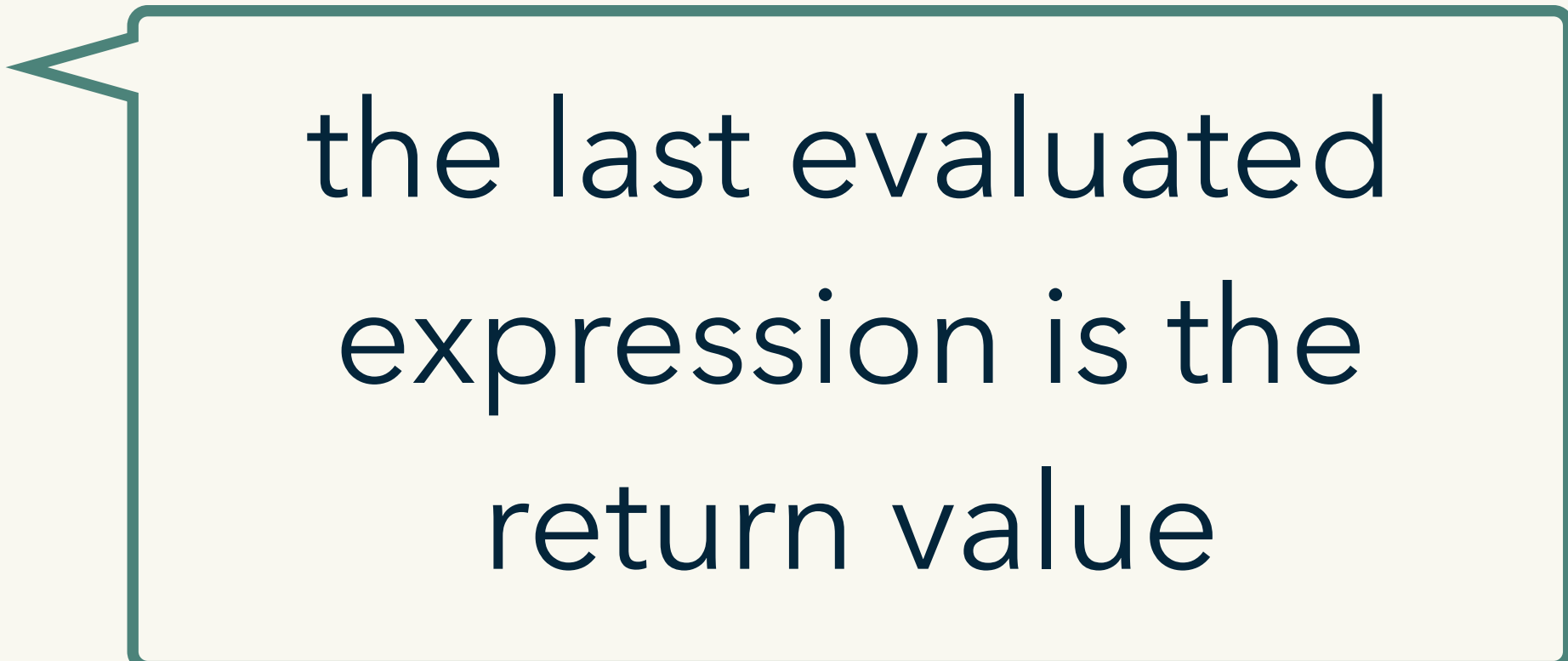
```
rescale01 <- function(x) {  
  (x - min(x)) / (max(x) - min(x))  
}
```

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])  
}
```



the last evaluated
expression is the
return value

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

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

Still some repetition,
we'll solve that later.

Why create a function? Because a function:

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

Write a function to remove the duplication

```
df <- tibble::tibble(  
  a = c(rnorm(9), -99),  
  b = c(-999, -99, rnorm(8)),  
  c = c(0, rnorm(9)),  
  d = rnorm(10)  
)
```

```
# Fix missing values
```

```
df$a[df$a == -99] <- NA  
df$b[df$b == -99] <- NA  
df$c[df$c == -99] <- NA  
df$d[df$d == -99] <- NA
```

Write a function to remove the duplication

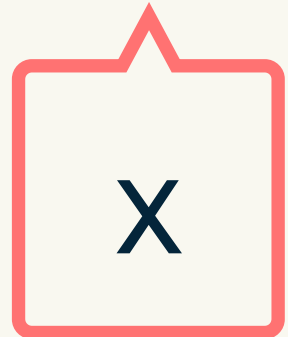
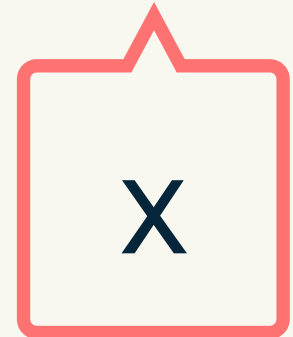
```
# Fix missing values
```

```
df$a[df$a == -99] <- NA
```

```
df$b[df$b == -99] <- NA
```

```
df$c[df$c == -99] <- NA
```

```
df$d[df$d == -99] <- NA
```



```
fix_missing <- function(x) {
```

```
}
```

Write a function to remove the duplication

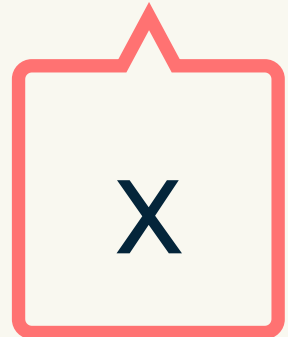
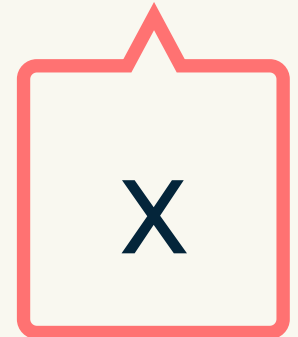
```
# Fix missing values
```

```
df$a[df$a == -99] <- NA
```

```
df$b[df$b == -99] <- NA
```

```
df$c[df$c == -99] <- NA
```

```
df$d[df$d == -99] <- NA
```



```
fix_missing <- function(x) {  
  df$a[df$a == -99] <- NA  
}
```

Write a function to remove the duplication

```
# Fix missing values
```

```
df$a[df$a == -99] <- NA
```

```
df$b[df$b == -99] <- NA
```

```
df$c[df$c == -99] <- NA
```

```
df$d[df$d == -99] <- NA
```

x

x

```
fix_missing <- function(x) {
```

```
  x[x == -99] <- NA
```

```
}
```

This expression doesn't return a value

```
fix_missing(c(0, -99, 2)) # nothing comes out!
```

Write a function to remove the duplication

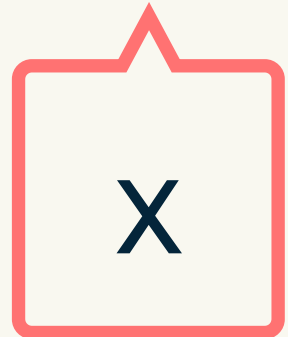
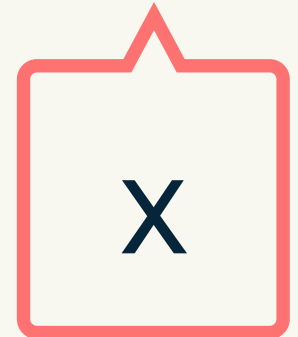
```
# Fix missing values
```

```
df$a[df$a == -99] <- NA
```

```
df$b[df$b == -99] <- NA
```

```
df$c[df$c == -99] <- NA
```

```
df$d[df$d == -99] <- NA
```



```
fix_missing <- function(x) {
```

```
  x[x == -99] <- NA
```

```
  x
```

```
}
```

```
fix_missing(c(0, -99, 2)) # Fixed!
```

How can we extend our function?

Rescale to [0, 1]

$0 + (1 - 0) * ((df\$a - \min(df\$a)) / (\max(df\$a) - \min(df\$a)))$

Rescale to [-1, 1]

$-1 + (1 - -1) * ((df\$b - \min(df\$b)) / (\max(df\$b) - \min(df\$b)))$

Rescale to [0, 10]

$0 + (10 - 0) * ((df\$c - \min(df\$c)) / (\max(df\$c) - \min(df\$c)))$

Identify the parts that might change,

```
# Rescale to [0, 1]
0 + (1 - 0) * ((df$a - min(df$a)) / (max(df$a) - min(df$a)))

# Rescale to [-1, 1]
-1 + (1 - -1) * ((df$b - min(df$b)) / (max(df$b) - min(df$b)))

# Rescale to [0, 10]
0 + (10 - 0) * ((df$c - min(df$c)) / (max(df$c) - min(df$c)))
```

And give them names

```
# Rescale to [0, 1]
```

```
0 + (1 - 0) * ((df$a - min(df$a)) / (max(df$a) - min(df$a)))
```

```
# Rescale to [-1, 1]
```

```
-1 + (1 - -1) * ((df$b - min(df$b)) / (max(df$b) - min(df$b)))
```

```
# Rescale to [0, 10]
```

```
0 + (10 - 0) * ((df$c - min(df$c)) / (max(df$c) - min(df$c)))
```

min

max

min

x

Starting from our earlier function...

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

Add new arguments,

```
rescale <- function(x, min, max) {  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```

And give them default values

```
rescale <- function(x, min = 0, max = 1) {  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```

How can we extend our function?

```
# Rescale to [0, 1]  
rescale(df$a)
```

```
# Rescale to [-1, 1]  
rescale(df$b, min = -1, max = 1)
```

```
# Rescale to [0, 10]  
rescale(df$c, max = 10)
```

Extend to allow for different codes for missing

```
fix_missing <- function(x) {  
  x[x == -99] <- NA  
  x  
}
```

```
# Fix missing values  
df$a[df$a == -99] <- NA  
df$b[df$b == -999] <- NA  
df$c[df$c == -0] <- NA
```

Extend to allow for different codes for missing

```
fix_missing <- function(x, missing_val = -99) {  
  x[x == missing_val] <- NA  
  x  
}
```

```
# Fix missing values
```

```
fix_missing(df$a)
```

```
fix_missing(df$b, missing_val = -9999)
```

```
fix_missing(df$c, missing_val = 0)
```

Debugging

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

To track down what is happening inside a function,

```
# This seems wrong...
```

```
rescale(df$a, c(0, 1))
```

```
# [1] 0.9910347 1.00000000 0.9929693 1.00000000
```

```
# [5] 0.9785413 1.00000000 0.9881966 1.00000000
```

```
# [9] 0.9754588 1.00000000
```


Use `debugonce()` to invoke the debugger

```
# Tells R to enter debugger, when rescale is called  
debugonce(rescale)
```

```
# Call the problem case  
rescale(df$a, c(0, 1))
```

Code about to be evaluated

Environment
as the
function sees
it.

Enter, to evaluate code

~/Documents/Projects/advanced-r - master - RStudio

Go to file/function Addins advanced-r

rescale x

Function: rescale (.GlobalEnv) (Read-only)

⚠ Debug location is approximate because the source is not available.

```
1 function(x, min = 0, max = 1) {
2   rng <- range(x, na.rm = TRUE, finite = TRUE)
3   min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))
4 }
```

Environment History Connections Jobs Git

Import Dataset List

rescale()

Values

max	1
min	num [1:2] 0 1
rng	num [1:2] -99 1.84
x	num [1:10] 1.84 0.185 0.424 0.13...

Traceback ☐ Show internals

rescale(df\$a, c(0, 1))

Files Plots Packages Help Viewer

Zoom Export

Console Terminal

~/Documents/Projects/advanced-r/

Next { } = Continue Stop

```
Browse[2]>
debug at #2: rng <- range(x, na.rm = TRUE, finite = TRUE)
Browse[2]>
debug at #3: min + (max - min) * ((x - rng[1]) / (rng[2] - rng
[1]))
Browse[2]> rng
[1] -99.000000 1.839591
Browse[2]> (max - min)
[1] 1 0
Browse[2]>
```

Write code to explore

~/Documents/Projects/advanced-r - master - RStudio

Go to file/function Addins advanced-r

rescale x

Function: rescale (.GlobalEnv) (Read-only)

⚠ Debug location is approximate because the source is not available.

```
1- function(x, min = 0, max = 1) {  
2-   rng <- range(x, na.rm = TRUE, finite = TRUE)  
→ 3-   min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
4- }
```

Environment History Connections Jobs Git

rescale()

Values

max	1
min	num [1:2] 0 1
rng	num [1:2] -99 1.84
x	num [1:10] 1.84 0.185 0.424 0.13...

Traceback ☐ Show internals

→ rescale(df\$a, c(0, 1))

Files Plots Packages Help Viewer

Console Terminal x

~/Documents/Projects/advanced-r/

Next { } ⏪ ⏩ Continue Stop

Stop, or

```
Browse[2]>  
debug at #2: rng <- range(x, na.rm = TRUE, finite = TRUE)  
Browse[2]>  
debug at #3: min + (max - min) * ((x - rng[1]) / (rng[2] - rng  
[1]))  
Browse[2]> rng  
[1] -99.000000 1.839591  
Browse[2]> (max - min)  
[1] 1 0  
Browse[2]> Q
```

Q + Enter

In this case, it was user error...

Not designed to take a
length 2 vector

This seems wrong...

```
rescale(df$a, c(0, 1))
```

```
# [1] 0.9910347 1.0000000 0.9929693 1.0000000
```

```
# [5] 0.9785413 1.0000000 0.9881966 1.0000000
```

```
# [9] 0.9754588 1.0000000
```

Signaling an error

To stop a function early, combine `if()` and `stop()`

```
rescale <- function(x, min = 0, max = 1) {  
  if () {  
    stop()  
  }  
  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```

Check for a logical condition

```
rescale <- function(x, min = 0, max = 1) {  
  if (length(min) != 1) {  
    stop()  
  }  
  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```


stop() with an informative message

```
rescale <- function(x, min = 0, max = 1) {  
  if (length(min) != 1) {  
    stop("`min` must have length 1", call. = FALSE)  
  }  
  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```

```
rescale(df$a, c(0, 1))  
# Error: `min` must have length 1
```

What else about the inputs should be checked?

```
rescale <- function(x, min = 0, max = 1) {  
  if (length(min) != 1) {  
    stop("`min` must have length 1", call. = FALSE)  
  }  
  
  rng <- range(x, na.rm = TRUE, finite = TRUE)  
  min + (max - min) * ((x - rng[1]) / (rng[2] - rng[1]))  
}
```

Brainstorm with your neighbour, then implement one more check.

min and max should be single numbers

min is smaller than max

x is also a vector of numbers

Functions are for
humans too

A.K.A API design

Principle:

Design your functions with *a user*
in mind.

Principle:

Design your functions with **future you**
in mind.

Case study

String manipulation

What makes base R functions hard to learn?

```
strsplit(x, split, ...)  
grep(pattern, x, value = FALSE, ...)  
grepl(pattern, x, ...)  
sub(pattern, replacement, x, ...)  
gsub(pattern, replacement, x, ...)  
regexpr(pattern, text, ...)  
gregexpr(pattern, text, ...)  
regexec(pattern, text, ...)  
substr(x, start, stop)  
nchar(x, type, ...)
```

A few issues


Names: Function names have no common theme, and no common prefix. Names are concise at expense of expressiveness.

Arguments: Argument names & order are not consistent, and data isn't the first argument. Sometimes text, sometimes x.

Type stability: `grep()` is not type stable: can return string or integer. Can't feed output of `gregexpr()` into `substr()`

Back at 10:35am

1. Carefully contemplate names
2. Plan for pipes
3. Keep it simple



Each individual
problem is small

Carefully
contemplate names

"A rose by any other name
would smell as sweet."
– *Shakespeare*



"A **function** by any other name
would **not** smell as sweet."
– *Hadley Wickham*



Principle:

Whenever you can give
something an informative name,
you should

stringr uses evocative verbs

`str_split()`

`str_detect()`

`str_locate()`

`str_subset()`

`str_extract()`

`str_replace()`

But good verbs don't always exist

`str_to_lower()`

`str_to_upper()`

stringr uses evocative verbs

```
str_split()  
str_detect()  
str_locate()  
str_subset()  
str_extract()  
str_replace()
```

A common prefix may be
useful to group related
functions together

But good verbs don't always exist

```
str_to_lower()  
str_to_upper()
```

Avoid verbs with dual meanings

filter()

weather()

cleave()

General advice

Be consistent!

Function names should be generally be verbs.

Prefer specific to general; concrete to abstract.

Avoid short names; err on the side of expressiveness.

Avoid names that differ in UK/US dialects.

Avoid names used in base R, or by similar packages.

You might get it wrong the first time

Your turn

Brainstorm names for these functions:

```
f <- function(x){  
  mean(is.na(x))  
}
```

```
h <- function(x){  
  length(unique(x))  
}
```

```
g <- function(x, y){  
  ifelse(x > y, x, y)  
}
```

Plan for pipes

Why is the pipe useful?

```
library(dplyr)
library(nycflights13)
by_dest <- group_by(flights, dest)
dest_delay <- summarise(by_dest,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
big_dest <- filter(dest_delay, n > 100)
arrange(big_dest, desc(delay))
```

But naming is hard work

```
foo <- group_by(flights, dest)
foo <- summarise(foo,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo <- filter(foo, n > 100)
arrange(foo, desc(delay))
```


But naming is hard work

```
foo1 <- group_by(flights, dest)
foo2 <- summarise(foo1,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo3 <- filter(foo2, n > 100)
arrange(foo2, desc(delay))
```

Alternatively, you could nest function calls

```
arrange(  
  filter(  
    summarise(  
      group_by(flights, dest),  
      delay = mean(dep_delay, na.rm = TRUE),  
      n = n()  
    ),  
    n > 100  
  ),  
  desc(delay)  
)
```

magrittr provides a third option

%>%

No intermediaries; read from left-to-right

```
flights %>%  
  group_by(dest) %>%  
  summarise(  
    delay = mean(dep_delay, na.rm = TRUE),  
    n = n()  
  ) %>%  
  filter(n > 100) %>%  
  arrange(desc(delay))
```

	Read left-to-right	Can omit intermediate names	Non-linear
$y \leftarrow f(x)$ $g(y)$	✓		✓
$g(f(x))$		✓	✓
$x \rightarrow$ $f() \rightarrow$ $g()$	✓	✓	

Principle:

Data arguments should come first

Most arguments fall in one of two classes

Data

Required

Core data

Often vectorised

Often called x or data

Details

Optional

Additional options

Scalar

Names are important

```
# Typically you can omit the names of the
```

```
# data arguments
```

```
ggplot(mtcars, aes(x = disp, y = cyl))
```

```
ggplot(data = mtcars, mapping = aes(...))
```

```
# Typically you shouldn't omit the names of
```

```
# of the details argument
```

```
mean(1:10, , TRUE)
```

```
mean(1:10, na.rm = TRUE)
```


Your turn

Which are the data arguments in `grepl()`?

Which are the details?

Which are the data arguments in `strsplit()`?

Which are the details?

Which are the data arguments in `merge()`?

Which are the details?

Keep it simple

Principle:

Aim for functions that do one thing, on the simplest object possible.

Rely on other tools to combine them in complex ways.

stringr functions work with character vectors

```
# So you can apply them to columns in a data frame with dplyr  
storms %>%
```

```
  mutate(first_letter = str_sub(name, 1, 1))
```

```
# You can apply them consecutively with %>%
```

```
c("cwickham@gmail.com") %>%  
  str_replace("@", " at ") %>%  
  str_replace("\\.", " dot ")
```

```
# You can iterate with purrr
```

```
sentences %>%  
  str_split(" ") %>%  
  map_chr(str_c, collapse = "")
```

Case Study

Switch to project
case_study

Your Turn

Examine the code in 01-report.R

Discuss with your neighbour:

1. What are the sources of repetition?
2. How would you describe the steps in the analysis?
3. What functions might you write? What would you call them?

What are the sources of repetition?

Whole analysis is repeated three times - once for each state: BC, OR, WA

Some steps are also repetitive, e.g. save plots for both PDF and PNG

Might imagine further repetition if we also need to do this for other variables in the data, i.e
total_dollar_amount, n_existing_customer

How would you describe the steps in the analysis?

1. Check file isn't too old (over 30 days from today)
2. Import data
3. Create a weekly summary
4. Plot weekly summary
5. Output plots

What functions might you write? What would you call them?

1. `check_not_outdated(file_path)`

2. Import data

3. `summarise_weekly(data)`

4. `plot_weekly(data)`

5. `ggsave_multiple(plot,
 exts = c(".pdf", ".png"))`

Challenge

Take a stab at writing `check_not_outdated()`

```
or_file_date <- or_file_path %>%  
  str_extract("[0-9]{4}-[0-9]{2}-[0-9]{2}") %>%  
  parse_date()
```

```
or_days_old <- difftime(lubridate::today(),  
  or_file_date,  
  units = "days")
```

```
or_days_old > 30
```

One solution

```
check_not_outdated <- function(path){  
  date <- path %>%  
    str_extract("[0-9]{4}-[0-9]{2}-[0-9]{2}") %>%  
    parse_date()  
  
  age <- difftime(lubridate::today(),  
    date,  
    units = "days")  
  
  age > 30  
}  
check_not_outdated(or_file_path)
```

Might break into simpler functions

```
file_date <- function(path){  
  path %>%  
    str_extract("[0-9]{4}-[0-9]{2}-[0-9]{2}") %>%  
    parse_date()  
}
```

```
file_age <- function(path){  
  difftime(lubridate::today(), file_date(path), units = "days")  
}
```

```
check_not_outdated <- function(path, threshold_days = 30){  
  age <- file_age(path)  
  age > threshold_days  
}
```

```
check_not_outdated(or_file_path)
```

Might force an error if outdated

```
check_not_outdated <- function(path, threshold_days = 30){
  date <- path %>%
    str_extract("[0-9]{4}-[0-9]{2}-[0-9]{2}") %>%
    parse_date()

  age <- difftime(lubridate::today(),
    date,
    units = "days")

  old <- age > threshold_days

  if(any(old)){
    old_files <- paste(path[old], "is", age[old], "days old", collapse = ", \n* ")
    stop(paste("Some files are outdated: \n*", old_files), call. = FALSE)
  }
  message("No files are outdated")
  invisible(age)
}
check_not_outdated(or_file_path)
```

In this case, we get vectorization for free

```
files <- dir("data") %>% path_ext_remove()  
file_paths <- path("data", files, ext = "csv")
```

```
file_paths
```

```
# data/BC_2018-07-14.csv data/OR_2018-07-15.csv  
# data/WA_2018-07-18.csv
```

```
check_not_outdated(file_paths)
```

```
# No files are outdated
```

Import step

```
or <- read_csv(or_file_path)
bc <- read_csv(bc_file_path)
wa <- read_csv(wa_file_path)
```

Repetition can be removed by **functional programming**.

How might we write summarise_weekly()?

```
or_weekly <-  
  or %>%  
  mutate(week = lubridate::week(date)) %>%  
  group_by(type, week) %>%  
  summarise(  
    date = first(date),  
    n = sum(!is.na(n_sales)),  
    mean = mean(n_sales, na.rm = TRUE))
```

What is fragile about this function?

```
summarise_weekly <- function(data){  
  data %>%  
  mutate(week = lubridate::week(date)) %>%  
  group_by(type, week) %>%  
  summarise(  
    date = first(date),  
    n = sum(!is.na(n_sales)),  
    mean = mean(n_sales, na.rm = TRUE))  
}
```

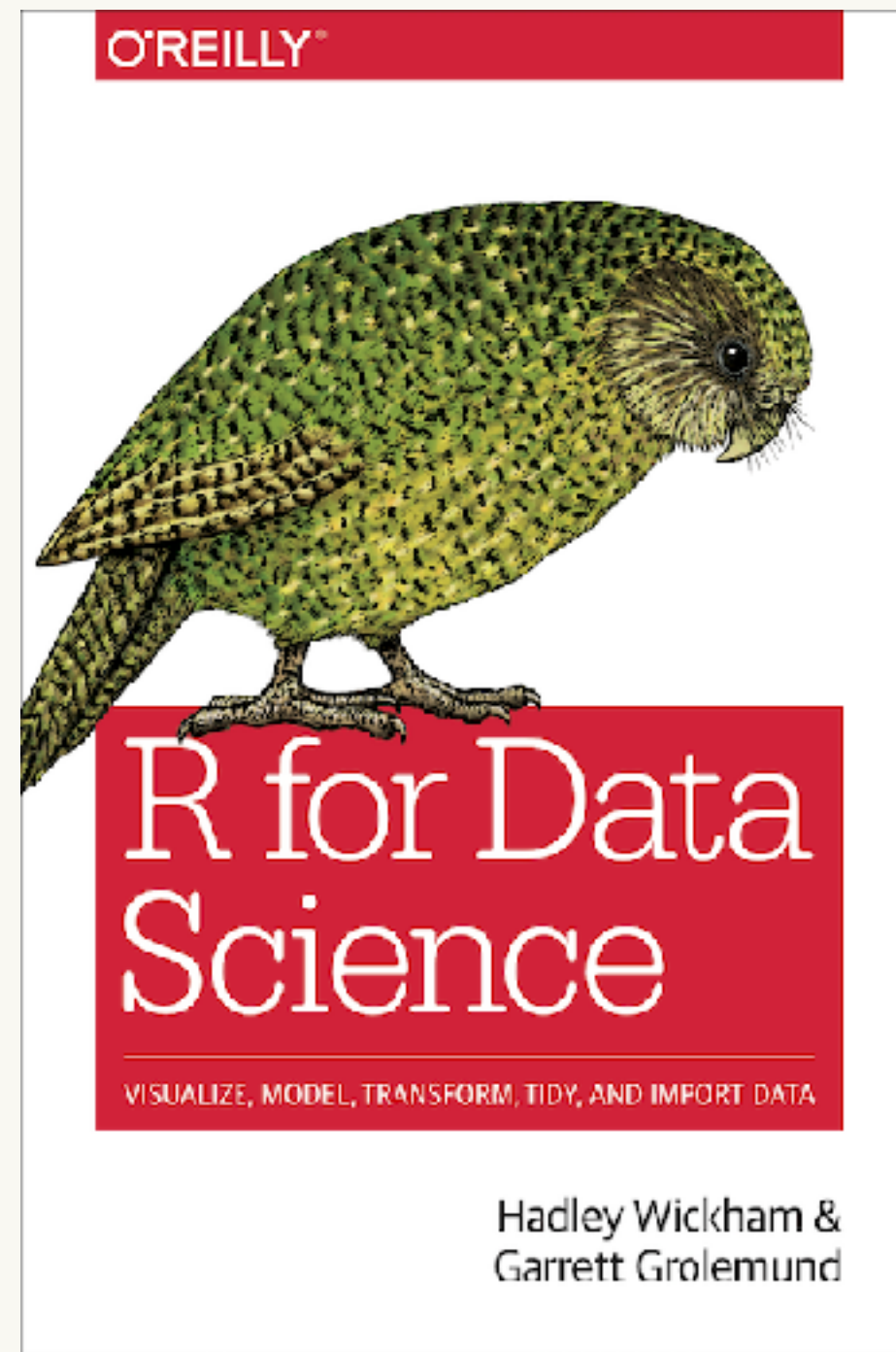
What is fragile about this function?

```
summarise_weekly <- function(data){  
  data %>%  
  mutate(week = lubridate::week(date)) %>%  
  group_by(type, week) %>%  
  summarise(  
    date = first(date),  
    n = sum(!is.na(n_sales)),  
    mean = mean(n_sales, na.rm = TRUE))  
}  
summarise_weekly(or)
```

Makes assumptions about
the contents of data.
Solve with **tidy evaluation**.

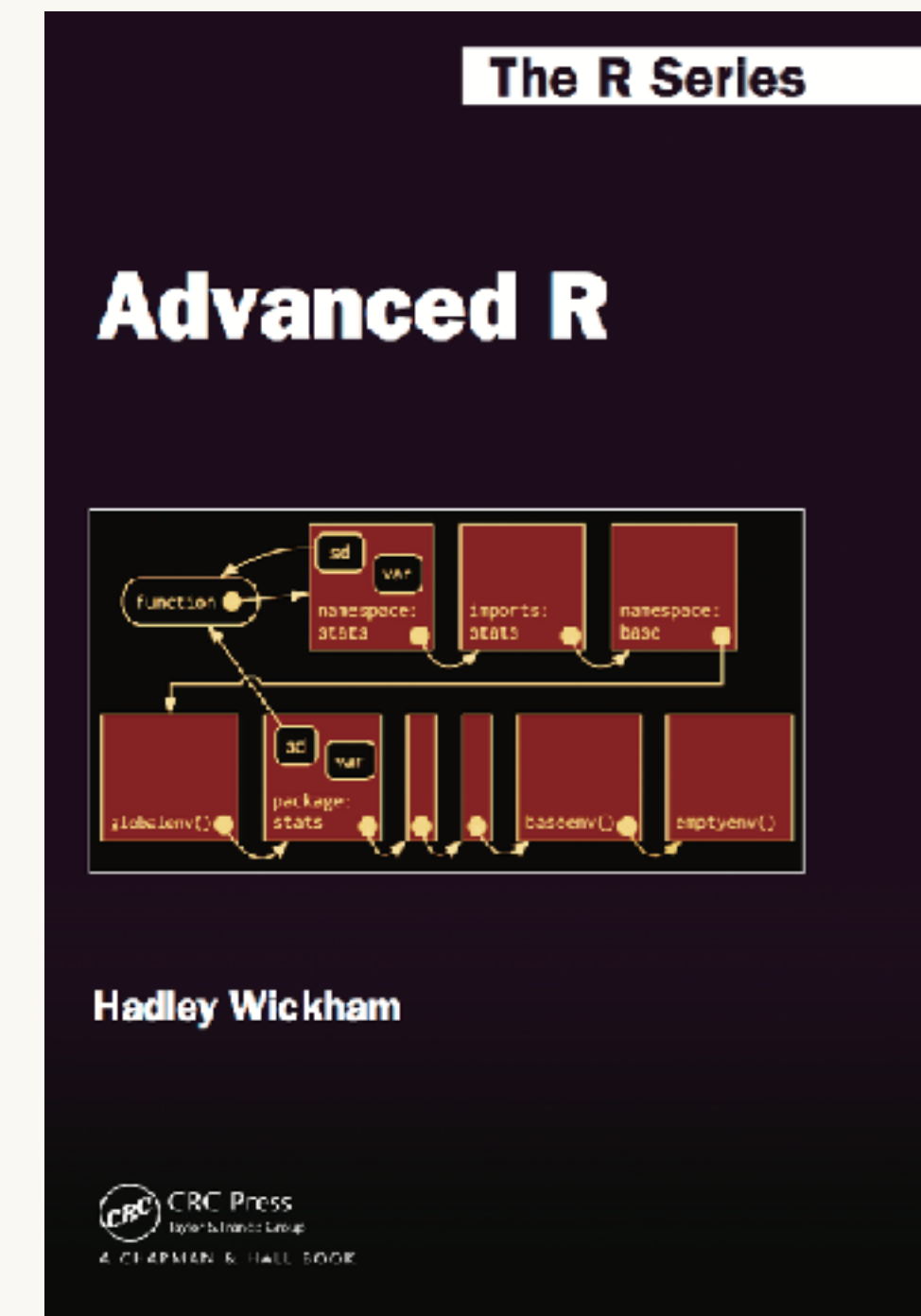
Learning more

Learning more



Functions

<http://r4ds.had.co.nz/functions.html>



Functions

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

Adapted from *Tidy Tools* by Hadley Wickham

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/](https://creativecommons.org/licenses/by-sa/4.0/)