Functional programming

July 2018

Charlotte Wickham
@cvwickham
cwickham@gmail.com
cwick.co.nz

Adapted from *Tidy Tools* by Hadley Wickham

Motivation

Copy and paste is a rich source of errors

```
# Fix missing values
df$a <- df$a[df$a == -99] <- NA
df$b <- df$b[df$b == -99] <- NA
df$c <- df$c[df$c == -99] <- NA
df$d <- df$d[df$d == -99] <- NA
df$e <- df$e[df$e == -99] <- NA
df$f <- df$f[df$f == -99] <- NA
df$g <- df$g[df$g == -98] <- NA
df$h <- df$h[df$h == -99] <- NA
df$i <- df$i[df$i == -99] <- NA
df$j <- df$i[df$j == -99] <- NA
df$k <- df$k[df$k == -99] <- NA
```

Copy and paste is a rich source of errors

```
# Fix missing values
df$a <- df$a[df$a == -99] <- NA
df$b <- df$b[df$b == -99] <- NA
df$c <- df$c[df$c == -99] <- NA
df$d <- df$d[df$d == -99] <- NA
df$e <- df$e[df$e == -99] <- NA
df$f <- df$f[df$f == -99] <- NA
df$g <- df$g[df$g == -98] <- NA
df$h <- df$h[df$h == -99] <- NA
df$i <- df$i[df$i == -99] <- NA
df$j <- df$i[df$j == -99] <- NA
df$k <- df$k[df$k == -99] <- NA
```

Functions can remove some sources of duplication

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$h <- fix_missing(df$i)
```

Functions can remove some sources of duplication

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$h <- fix_missing(df$i)
```

For loops can remove others

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

for (i in seq_along(df)) {
  df[[i]] <- fix_missing(df[[i]])
}
```

Why for loops are bad

Why for loops are ~~bad~~ suboptimal

What does this code do?

```
out1 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

out2 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

For loops emphasise the objects

```
out1 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

out2 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

Not the actions

```
out1 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out1[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}

out2 <- vector("double", ncol(mtcars))
for(i in seq_along(mtcars)) {
  out2[[i]] <- median(mtcars[[i]], na.rm = TRUE)
}
```

Functional programming emphasises the actions

```
library(purrr)

means <- map_dbl(mtcars, mean)
medians <- map_dbl(mtcars, median)
```

Read: for each element of mtcars, apply mean.

FP tools allow you to focus on what happens

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$i <- fix_missing(df$i)
```

FP tools allow you to focus on what happens

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

And provide useful tools for generalisation

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

Foundations

Warmup: Guess the output, then run to check.

```
# Guess the output
fun1 <- mean
fun1(c(1, 3))

# Guess the output
fun2 <- `+`
fun2(1, 3)
```

How can we remove the duplication here?

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

# Center columns
df$a <- df$a - mean(df$a)
df$b <- df$b - median(df$b)
df$c <- df$c - dplyr::first(df$c)
```

Your Turn

Write the function recenter() to remove the repetition:

```
df$a <- recenter(df$a, mean)
df$b <- recenter(df$b, median)
df$c <- recenter(df$c, dplyr::first)
```

Identify what might change

```
df$a - mean(df$a)
df$b - median(df$b)
df$c - dplyr::first(df$c)
```

Give them names,

```
  x              fun    x
df$a -         mean(df$a)
df$b -       median(df$b)
df$c - dplyr::first(df$c)
```

Make the template

```
recenter <- function(x, fun){

}
```

Copy in an example

```
recenter <- function(x, fun){
  df$a - mean(df$a)
}
```

And use the argument names

```
recenter <- function(x, fun){
  x - mean(x)
}
```

And use the argument names

```
recenter <- function(x, fun){
  x - fun(x)
}
```

And use the argument names

```
recenter <- function(x, fun){
  x - fun(x)
}

df$a <- recenter(df$a, mean)
df$b <- recenter(df$b, median)
df$c <- recenter(df$c, dplyr::first)
```

Passing in a function as an argument

Functions are **first class** citizens in R

Anything you can do with a vector you can do with a function, e.g.:

1. Functions can be assigned to variables.
2. Functions can be arguments to functions.
3. Functions can be returned from a function.
4. Functions can be stored in lists.

Functions are **first class** citizens in R

Anything you can do with a vector you can do with a function, e.g.:

1. Functions can be assigned to variables.
2. **Functions can be arguments to functions.**
3. Functions can be returned from a function.
4. Functions can be stored in lists.

Solving iteration problems with the map() family

A family of functionals

Map strategy

For an iteration task:

1. Solve for single .x
2. Generalise solution with appropriate map() function
3. Simplify (if possible)

Find first element of compound string

```r
x1 <- c("a|b", "a|b|c", "d|e", "b|c|d")
# We want:
# "a" "a" "d" "b"

# A useful intermediate object
x2 <- strsplit(x1, "|", fixed = TRUE)

# For each element of x2
# pull out the first element
```

```
# [[1]]
# [1] "a" "b"
#
# [[2]]
# [1] "a" "b" "c"
#
# [[3]]
# [1] "d" "e"
#
# [[4]]
# [1] "b" "c" "d"
```

1. Solve for single .x

```
# Pull out one element
.x <- x2[[1]]
```

Specially named pronoun that map understands

```
.x
# [1] "a" "b"

# Get first element
.x[[1]]
# Solved!
```

2. Generalise solution with map()

```
# Solution for one element
          .x[[1]]

# Turn into a recipe with ~ and pass to map
map(x2, ~ .x[[1]])
```

For each element of x2,

take it, and extract the first element

Your Turn

Compute the mean of every column in mtcars.

Generate a sample of size 10 from Normals with the following means: -10, 0, 10, 100

Compute the number of unique values in each column of iris

Back at 1:00pm

Compute the mean of every column in mtcars

```
# Solve for one
.x <- mtcars[[1]]
mean(.x)

# Generalise
map(mtcars, ~ mean(.x))
```

Generate 10 random normals

```r
mu <- c(-10, 0, 10, 100)

# Solve for one
.x <- mu[[1]]
rnorm(10, mean = .x)

# Generalise
map(mu, ~ rnorm(10, mean = .x))
```

Compute the number of unique values in each column

```
# Solve for one
.x <- iris[[1]]
length(unique(.x))

# Generalise
map(iris, ~ length(unique(.x)))
```

Map strategy

For an iteration task:

1. Solve for single .x
2. Generalise solution with appropriate map() function
3. Simplify (if possible)

Each variant always produces the same type

Function	Output
`map_lgl()`	Logical vector
`map_int()`	Integer vector
`map_dbl()`	Double vector
`map_chr()`	Character vector
`map()`	List
`map_dfc()`	Data frame (by col)
`map_dfr()`	Data frame (by row)

Map strategy

For an iteration task:

1. Solve for single .x
2. Generalise solution with appropriate map() function
3. Simplify (if possible)

Simplify extraction

```
map(z, ~ .x[[1]])
map(z, 1)

map(z, ~ .x[["string"]])
map(z, "string")

map(z, ~ .x[["string"]][[1]] %||% NA)
map(z, list("string", 1), .default = NA))
```

Simplify function calls

```
map(z, ~ f(.x))
map(z, f)

map(z, ~ f(.x, a = 1, b = 2))
map(z, f, a = 1, b = 2)

map(z, ~ f(first_arg = 1, .x))
map(z, f, first_arg = 1)
```

Your Turn

For the three examples earlier:

```
map(mtcars, ~ mean(.x))
map(mu, ~ rnorm(10, mean = .x))
map(iris, ~ length(unique(.x)))
```

Use an appropriate map function, and simplify if possible.

Compute the mean of every column in mtcars

```
# Appropriate function
map_dbl(mtcars, ~ mean(.x))

# Simplify (optional)
map_dbl(mtcars, mean)
```

Generate 10 random normals

```
mu <- c(-10, 0, 10, 100)

# Appropriate function?
map(mu, ~ rnorm(10, mean = .x))
map_dfc(mu, ~ rnorm(10, mean = .x)) #?

# Simplify (optional)
map(mu, rnorm, n = 10)
```

Compute the number of unique values in each column

```r
# Appropriate function
map_int(iris, ~ length(unique(.x)))

# Simplify ?
nunique <- function(x) length(unique(x))
map_int(iris, ~ nunique(.x))
map_int(iris, nunique)
```

Typo

FP tools allow you to focus on what happens

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$f)
df$g <- fix_missing(df$g)
df$h <- fix_missing(df$h)
df$i <- fix_missing(df$i)
```

Why isn't this quite what we want?

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

df2 <- map(df, fix_missing)

# Hint: look at the structure of df2
```

Why isn't this quite what we want?

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

df2 <- map(df, fix_missing)
str(df2)
# List of 4
# $ a: num [1:10] 10.53 10.32 9.87 10.5 8.07 ...
# $ b: num [1:10] -998.33 -98.33 0.419 -1.084 -0.258 ...
# $ c: num [1:10] 0 0.121 0.863 0.554 0.677 ...
# $ d: num [1:10] -0.769 1.325 1.972 -0.185 -0.289 ...
```

Why isn't this quite what we want?

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

df[] <- map(df, fix_missing)
str(df)
# Classes ‘tbl_df’, ‘tbl’ and 'data.frame': 10 obs. of  4 variables
# $ a: num  10.53 10.32 9.87 10.5 8.07 ...
# $ b: num  -998.33 -98.33 0.419 -1.084 -0.258 ...
# $ c: num  0 0.121 0.863 0.554 0.677 ...
# $ d: num  -0.769 1.325 1.972 -0.185 -0.289 ...
```

Why isn't this quite what we want?

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

df <- modify(df, fix_missing)
str(df)
# Classes ‘tbl_df’, ‘tbl’ and 'data.frame': 10 obs. of  4 variables
# $ a: num  10.53 10.32 9.87 10.5 8.07 ...
# $ b: num  -998.33 -98.33 0.419 -1.084 -0.258 ...
# $ c: num  0 0.121 0.863 0.554 0.677 ...
# $ d: num  -0.769 1.325 1.972 -0.185 -0.289 ...
```

Why not base R?

Compared to purrr, base R functions:

Have inconsistent names (lapply() vs. Map())

Have inconsistent argument order (lapply() vs. mapply())

Require functions (no 1, or extract helpers)

Are either type-unstable (sapply()) or verbose (vapply())

Lack side-effect form (no walk())

Lack paired maps (no map2())

Lack data frame output (no _dfc(), _dfr())

purrr provides a full set of functions

Number of inputs	Output is a **scalar**	Output is **anything**	Output is **nothing**
1	`map_lgl()`, `map_int()`, `map_dbl()`, `map_chr()`	`map()`	`walk()`
2	`map2_lgl()`, `map2_int()`, `map2_dbl()`, `map2_chr()`	`map2()`	`walk2()`
n	`pmap_lgl()`, `pmap_int()`, `pmap_dbl()`, `pmap_chr()`	`pmap()`	`pwalk()`

Base R only provides a partial set of functions

Number of inputs	Output is a **scalar**	Output is **anything**	Output is **nothing**
1	`sapply()` / `vapply()`	`lapply()`	
2			
n	`mapply()`	`Map()`	

Paired map

stringr application

```
# How do we go from locations to words?
# Easy if we have a single location
pos <- str_locate(sentences, "\\b\\w{5,}\\b")
str_sub(sentences, pos)

# NB: str_sub can take one
# 2 column matrix, or two vectors
```

A word with 5 or more letters.

What if we have multiple locations?

```
pos <- str_locate_all(
  sentences, "\\b\\w{5,}\\b"
)

# Solve for one instance: now have two inputs!
.x <- sentences[[1]]
.y <- pos[[1]]
```

Another special pronoun

```
str_sub(.x, .y)
```

Generalise & simplify

```
# Generalise
map2(sentences, pos, ~ str_sub(.x, .y))
```

1st input

2nd input

```
# Simplify
map2(sentences, pos, str_sub)
```

walk2() is often useful when writing files

```
diamonds <- ggplot2::diamonds
by_color <- split(diamonds, diamonds$color)
paths <- paste0(names(by_color), ".csv")

# Solve for one
.x <- by_color[[1]]
.y <- paths[[1]]
write.csv(.x, .y)

# Solve for all
walk2(by_color, paths, ~ write.csv(.x, .y))

# Simplify
walk2(by_color, paths, write.csv)
```

Principle:

Compose value functions with map();
compose effect functions with walk()

Your Turn

```
diamonds <- ggplot2::diamonds
by_color <- split(diamonds, diamonds$color)
plots <- map(by_color, ~ ggplot(.x, aes(clarity, price)) +
                           geom_point())
plot_paths <- paste0(names(by_color), ".pdf")
```

Save the plots to the corresponding paths.

Your Turn

```
# Solve for one
.x <- plots[[1]]
.y <- plot_paths[[1]]

ggsave(.y, .x)

# Generalise
walk2(plots, plot_paths, ~ ggsave(.y, .x))

# Simplify
walk2(plot_paths, plots, ggsave)
```

```
# To clean up
file.remove(paths)
file.remove(plot_paths)
```

Functions are **first class** citizens in R

Anything you can do with a vector you can do with a function, e.g.:

1. Functions can be assigned to variables.
2. Functions can be arguments to functions.
3. **Functions can be returned from a function.**
4. Functions can be stored in lists.

Handling errors with safely()

A functional operator

What happens when there is an error?

```
input <- list(1:10, sqrt(4), 5, "n")
map(input, log)
```

What does safely() do?

```
# safely() modifies a function so it never fails
safe_log <- safely(log)

# The output of safely() is a function
safe_log
# function (...)
# capture_error(.f(...), otherwise, quiet)
# <bytecode: 0x7fcb996c2398>
# <environment: 0x7fcb99679058>
```

What does safely() do?

```
# What does it return when the function succeeds?
safe_log(1:10)

# What does it return when the function fails?
safe_log("n")
```

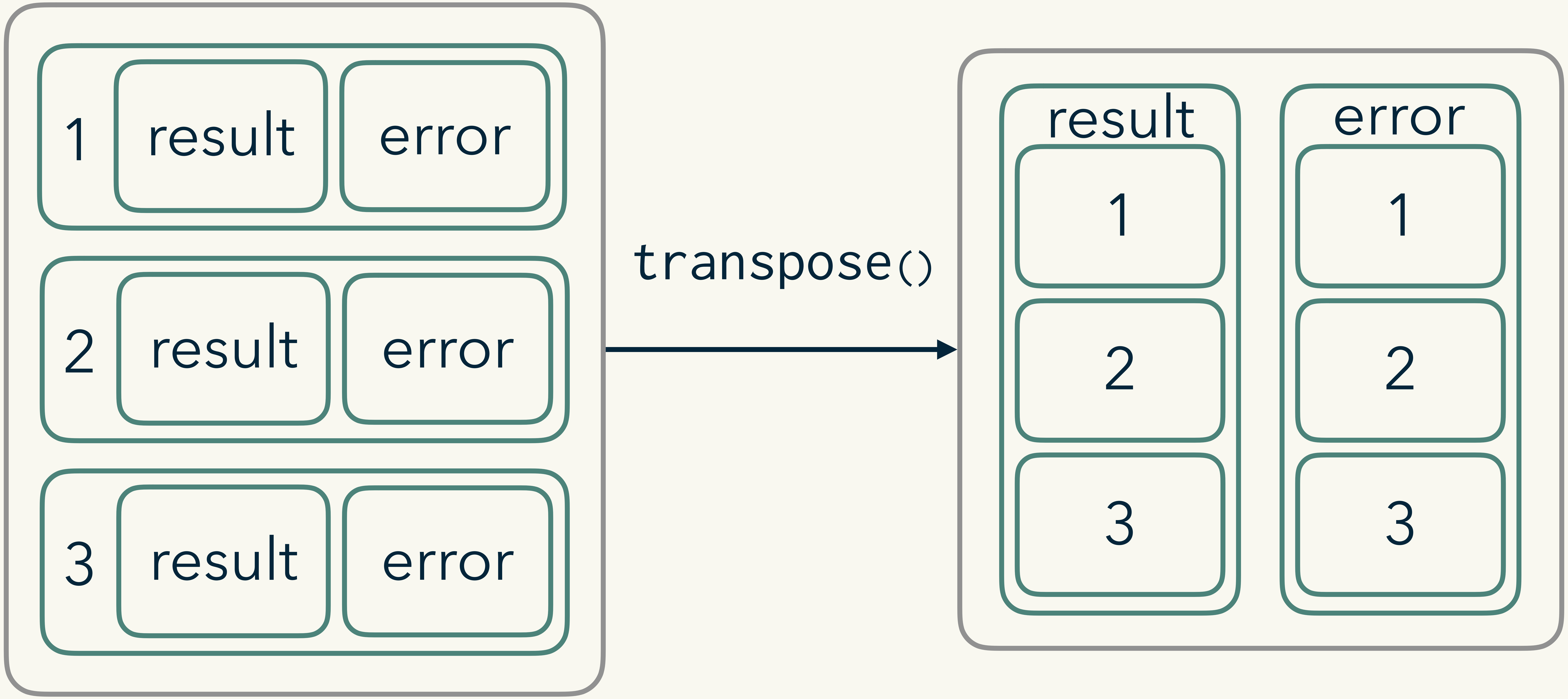
A more useful example

```
urls <- c(
  "https://google.com",
  "https://en.wikipedia.org",
  "asdfasdasdkfjlda"
)

# Fails
contents <- map(urls, readLines, warn = FALSE)

# Always succeeds
contents <- urls %>%
  map(safely(readLines), warn = FALSE)
str(contents)
```

But map() + safely() gives awkward output



Your turn

```
urls <- c(
  "https://google.com",
  "https://en.wikipedia.org",
  "asdfasdasdkfjlda"
)
contents <- urls %>%
  map(safely(readLines), warn = FALSE)
```

Apply transpose() to contents then:

1. Make logical vector that is TRUE if download succeeded. (Hint: use map_lgl())
2. List failed urls
3. Extract successfully retrieved text

Common pattern with safely()

```
contents <- urls %>%
  map(safely(readLines)) %>%
  transpose()

ok <- map_lgl(contents$error, is.null)
# This is suboptimal:
ok <- !map_lgl(contents$result, is.null)

urls[!ok]
contents$result[ok]
```

Case Study

Switch to project

case_study

Your Turn

Open 02-report.R.

Pay attention to contents of `files`, `file_paths` and `states`.

Remove the duplication in the data import step:

```
# Import data ----------------------------------------------

or <- read_csv(or_file_path)

bc <- read_csv(bc_file_path)

wa <- read_csv(wa_file_path)
```

One Solution

```
# Import data
---------------------------------------------------

all_states <- map(file_paths, read_csv)
```

Requires downstream edits too...

```
# Instead of this
or_weekly <- summarise_weekly(or)
bc_weekly <- summarise_weekly(bc)
wa_weekly <- summarise_weekly(wa)

# Will now need
all_states_weekly <- map(all_states, summarise_weekly)
```

Requires downstream edits too...

```
# Instead of this
or_plot <- plot_weekly(or_weekly, "Oregon")
bc_plot <- plot_weekly(bc_weekly, "British Columbia")
wa_plot <- plot_weekly(wa_weekly, "Washington")

# Will now need
all_states_plots <- map2(all_states_weekly,
  states_long_names,
  plot_weekly)
```

Your Turn

```
or_plot <- plot_weekly(or_weekly, "Oregon")
or_image_path <- path("images",
  paste(or_file, "n_sales", sep = "_"))

ggsave(paste0(or_image_path, ".pdf"), or_plot, height = 3, width = 8)
ggsave(paste0(or_image_path, ".png"), or_plot, height = 3, width = 8)
```

Use an appropriate map function to save or_plot to all extensions.

1. Solve for one extension

```
exts <- c(".pdf", ".png")

.x <- exts[[1]]

ggsave(paste0(or_image_path, .x),
  or_plot, height = 3, width = 8)
```

2. Generalize & Simplify

```
# Generalize
walk(exts, ~ ggsave(paste0(or_image_path, .x),
  or_plot, height = 3, width = 8))

# Simplify
walk(exts, ~ ggsave(paste0(or_image_path, .x)),
  plot = or_plot, height = 3, width = 8)

# Even better
image_paths <- paste0(or_image_path, exts)
walk(image_paths, ggsave,
  plot = or_plot, height = 3, width = 8)
```

Challenge

Try writing the ggsave_multiple() function:

```
exts <- c(".pdf", ".png")
image_paths <- paste0(or_image_path, exts)

walk(image_paths, ggsave,
  plot = or_plot, height = 3, width = 8)
```

1. What might vary? What are our argument names?

```
exts <- c(".pdf", ".png")
image_paths <- paste0(or_image_path, exts)

walk(image_paths, ggsave,
  plot = or_plot, height = 3, width = 8)
```

1. What might vary? What are our argument names?

```
exts <- c(".pdf", ".png")
image_paths <- paste0(or_image_path, exts)

walk(image_paths, ggsave,
  plot = or_plot, height = 3, width = 8)
```

exts

filename

plot

any other args to ggsave

Put in function template, matching ggsave() arg order

```
ggsave_multiple <- function(filename, plot, exts, ...){
  paths <- paste0(filename, exts)

  walk(paths, ggsave,
    plot = plot, ...)
}
ggsave_multiple(or_image_path, or_plot, c(".pdf", ".png"),
  height = 3, width = 8)
```

Downstream changes

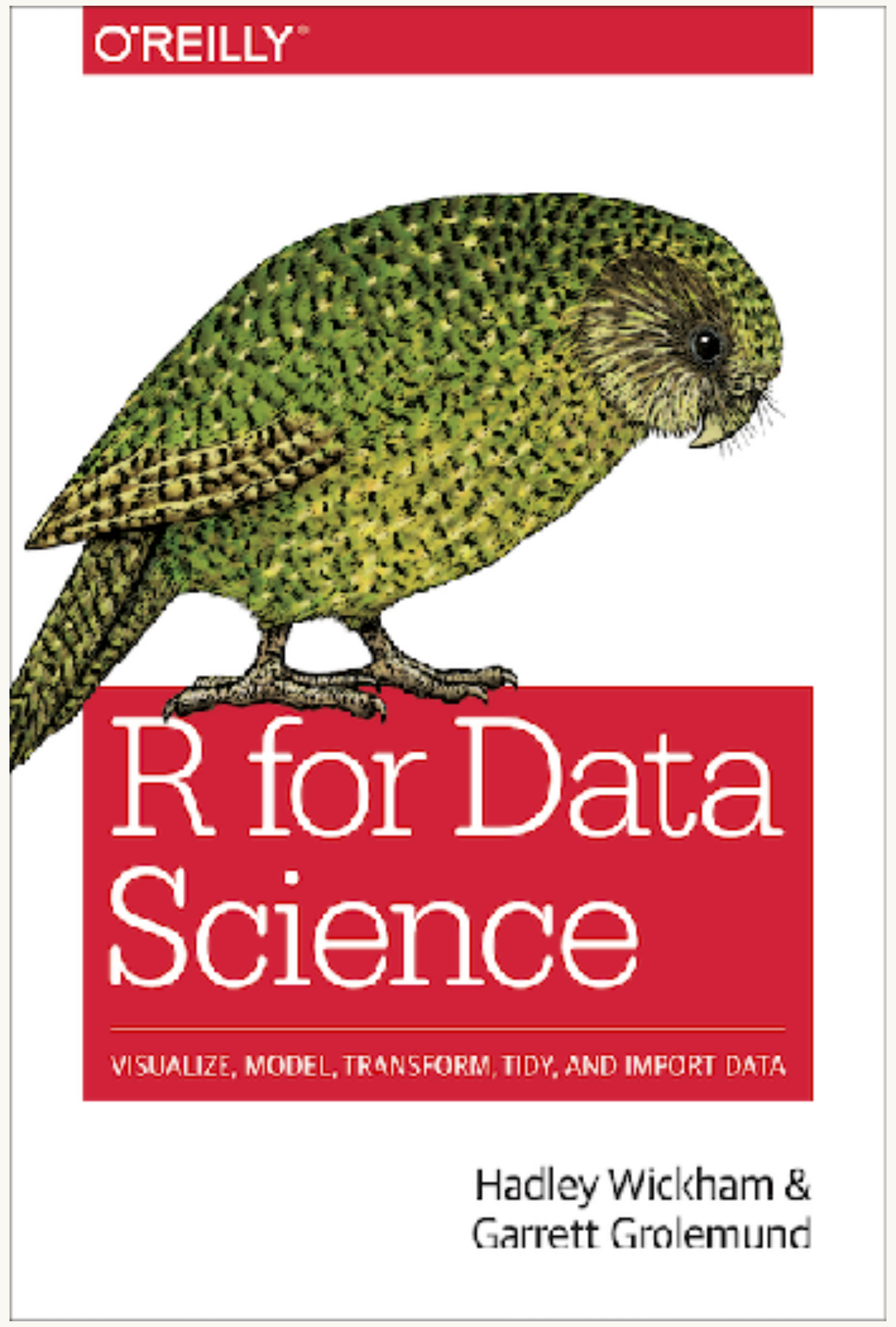
```
# This
ggsave_multiple(or_image_path, or_plot, c(".pdf", ".png"),
  height = 3, width = 8)
ggsave_multiple(bc_image_path, bc_plot, c(".pdf", ".png"),
  height = 3, width = 8)
ggsave_multiple(wa_image_path, wa_plot, c(".pdf", ".png"),
  height = 3, width = 8)

# Becomes
image_paths <- path("images",
  paste(files, "n_sales", sep = "_"))

walk2(image_paths, all_states_plots, ggsave_multiple,
  exts = c(".pdf", ".png"), height = 3, width = 8)
```

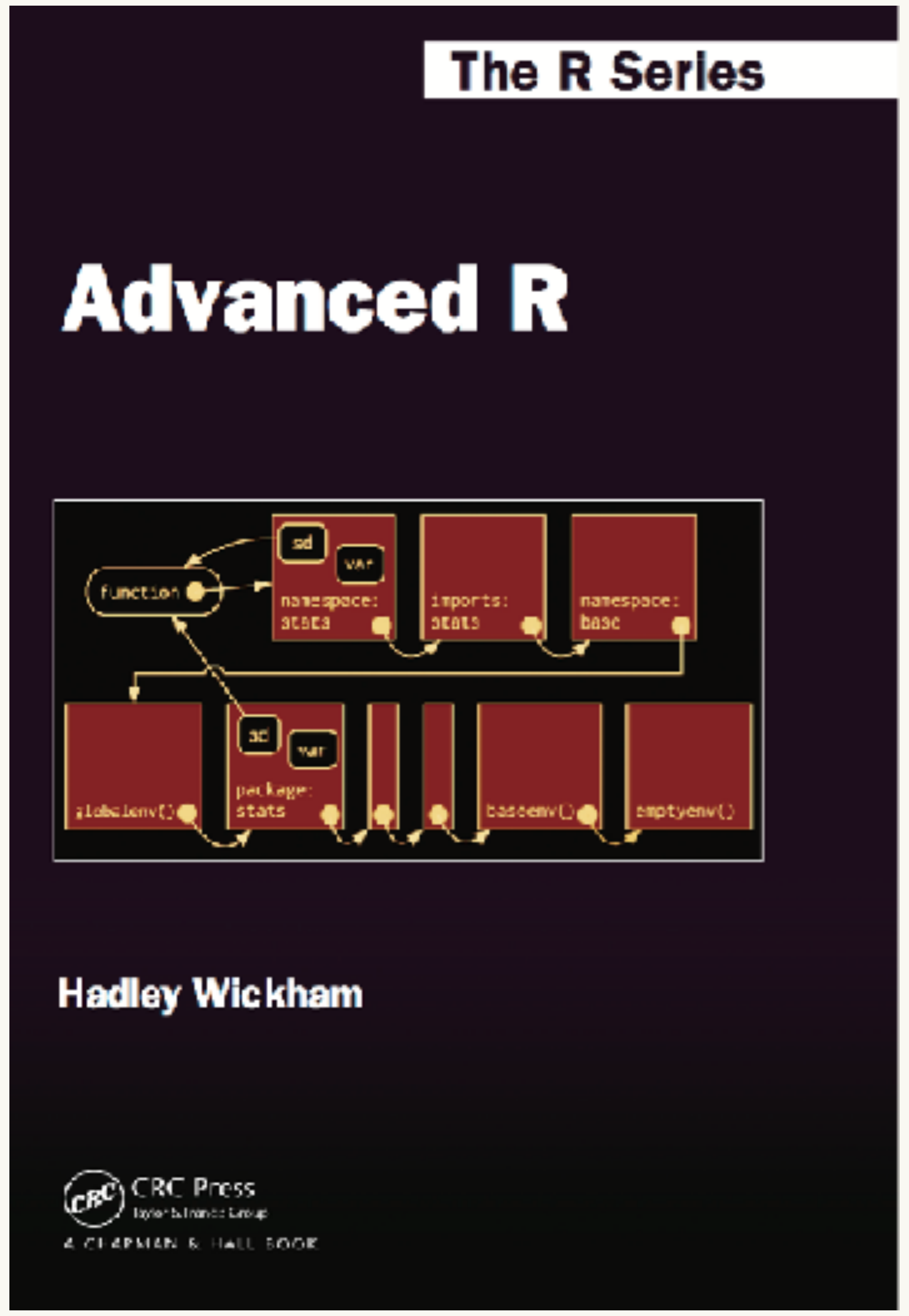
Learning more

Learning more



Iteration

http://r4ds.had.co.nz/iteration.html



Functional Programming

https://adv-r.hadley.nz/functional-programming.html