Functional programming

January 2018

Hadley Wickham
@hadleywickham
Chief Scientist, RStudio

Motivation

Copy and paste is a rich source of errors

```
# 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
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
df$k[df$k == -99] <- NA
```

Copy and paste is a rich source of errors

```
# 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
df$e[df$e == -99] <- NA
df$f[df$f == -99] <- NA
df$g[df$g == -98] <- NA
df$h[df$h == -99] <- NA
df$i[df$i == -99] <- NA
df$i[df$j == -99] <- NA
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

A detour with cupcakes

Why for loops are ~~bad~~ suboptimal

A detour with cupcakes

Vanilla cupcakes

The hummingbird bakery cookbook

1 cup flour
a scant ¾ cup sugar
1 ½ t baking powder
3 T unsalted butter
½ cup whole milk
1 egg
¼ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Chocolate cupcakes

¾ cup + 2T flour
2 ½ T cocoa powder
a scant ¾ cup sugar
1 ½ t baking powder
3 T unsalted butter
½ cup whole milk
1 egg
¼ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, cocoa, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Chocolate cupcakes

The hummingbird bakery cookbook

¾ cup + 2T flour
2 ½ T cocoa powder
a scant ¾ cup sugar
1 ½ t baking powder
3 T unsalted butter
½ cup whole milk
1 egg
¼ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, cocoa, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Vanilla cupcakes

1 cup flour
a scant ¾ cup sugar
1 ½ t baking powder
3 T unsalted butter
½ cup whole milk
1 egg
¼ t pure vanilla extract

Preheat oven to 350°F.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

Vanilla cupcakes

The hummingbird bakery cookbook

120g flour
140g sugar
1.5 t baking powder
40g unsalted butter
120ml milk
1 egg
0.25 t pure vanilla extract

Preheat oven to 170°C.

Put the flour, sugar, baking powder, salt, and butter in a freestanding electric mixer with a paddle attachment and beat on slow speed until you get a sandy consistency and everything is combined.

Whisk the milk, egg, and vanilla together in a pitcher, then slowly pour about half into the flour mixture, beat to combine, and turn the mixer up to high speed to get rid of any lumps.

Turn the mixer down to a slower speed and slowly pour in the remaining milk mixture. Continue mixing for a couple of more minutes until the batter is smooth but do not overmix.

Spoon the batter into paper cases until 2/3 full and bake in the preheated oven for 20-25 minutes, or until the cake bounces back when touched.

1. Convert units

Vanilla cupcakes

The hummingbird bakery cookbook

120g flour
140g sugar
1.5 t baking powder
40g butter
120ml milk
1 egg
0.25 t vanilla

Beat flour, sugar, baking powder, salt, and butter until sandy.

Whisk milk, egg, and vanilla. Mix half into flour mixture until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

2. Rely on domain knowledge

Vanilla cupcakes

The hummingbird bakery cookbook

120g flour
140g sugar
1.5 t baking powder
40g butter
120ml milk
1 egg
0.25 t vanilla

Beat dry ingredients + butter until sandy.

Whisk together wet ingredients. Mix half into dry until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

3. Use variables

Cupcakes

Beat dry ingredients + butter until sandy.

Whisk together wet ingredients. Mix half into dry until smooth (use high speed). Beat in remaining half. Mix until smooth.

Bake 20-25 min at 170°C.

Vanilla	Chocolate
120g flour	100g flour
	20g cocoa
140g sugar	140g sugar
1.5t baking powder	1.5t baking powder
40g butter	40g butter
120ml milk	120ml milk
1 egg	1 egg
0.25 t vanilla	0.25 t vanilla

4. Extract out common code

What do these for loops 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)
```

And back...

For loops can remove others

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

for (i in seq_along(df)) {
  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, is.numeric, fix_missing)
```

Principle:

Solve a single problem

Principle:
Scale up with map & friends

Warmups

What is `NA_real_`? `NA_integer_`? `NA_character_`?

Why don't you normally need to care?

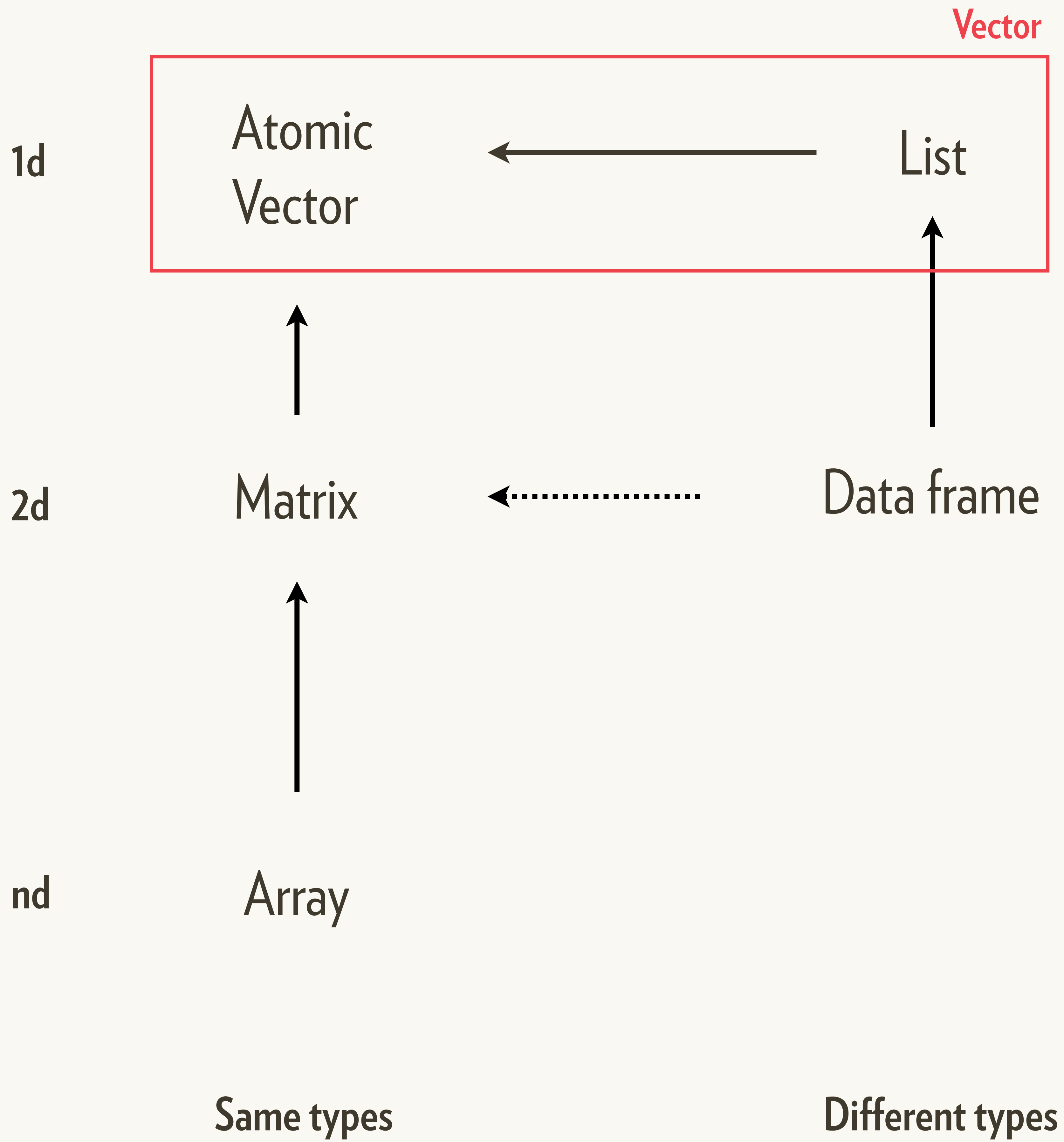
```r
# One NA for each basic atomic vector
typeof(NA)
typeof(NA_real_)
typeof(NA_integer_)
typeof(NA_character_)

c(NA, "x")
```

Your turn

How is a list different from an atomic vector?

How is a data frame different from a matrix?

How do you examine the structure of an object?



http://adv-r.had.co.nz/Data-structures.html `str()`

`View()`

(If you have RStudio 1.1)

Your turn

What's the difference between [and [[?

	Single	Multiple
Vectors	`x[[1]]`	`x[1:4]`
Lists	`x[[1]]` `x$name`	`x[1]`

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

What does this code do?

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

Change project to:

[colsum]

This package automatically loads purrr

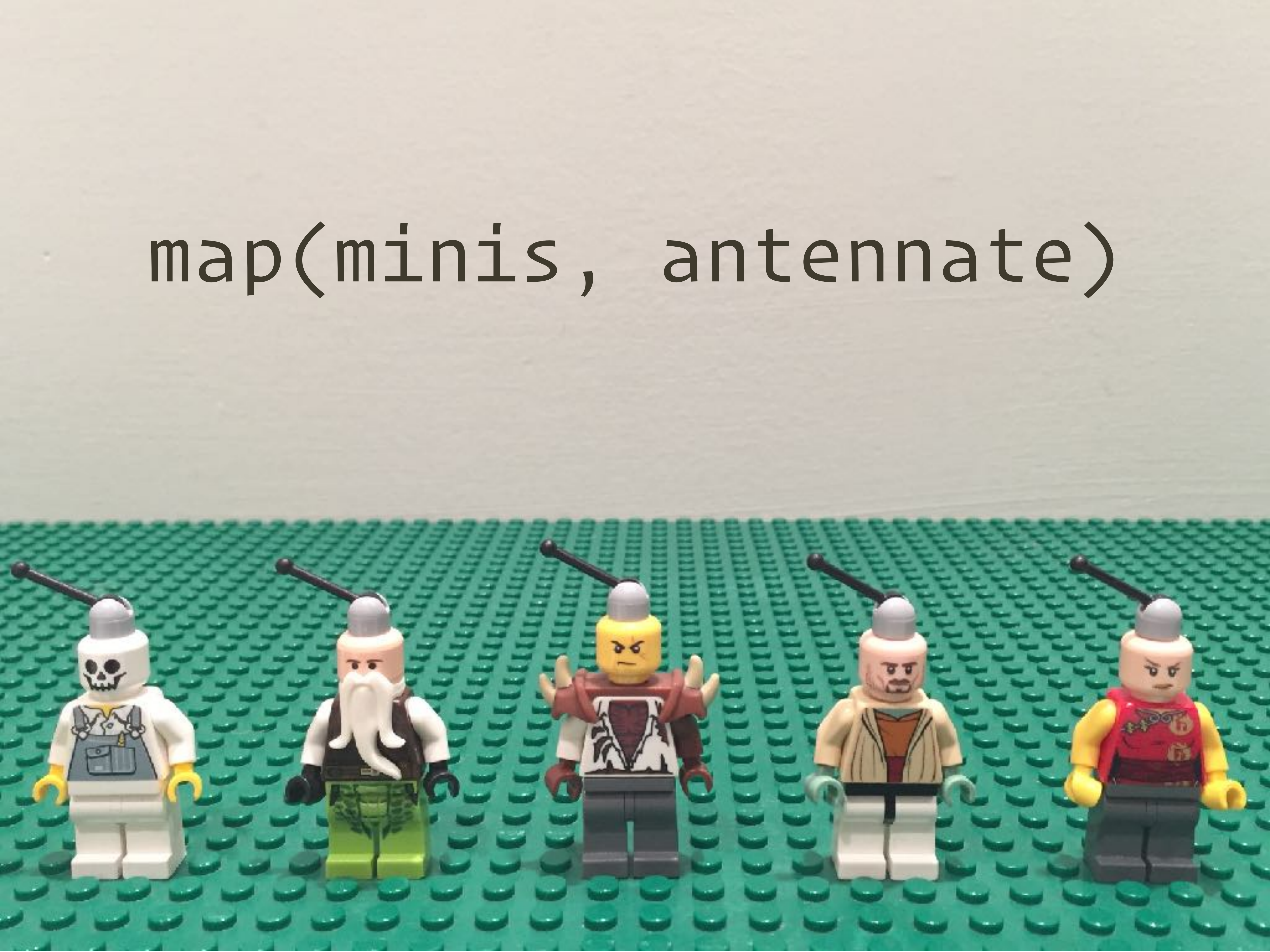
```
devtools::load_all(".")
Loading colsum
Loading required package: purrr
Attaching package: 'purrr'
```

```
# Because earlier I ran
use_package("purrr", "depends")
```

Pros	Cons
Easily call purrr functions	Affects global search path
	Not acceptable on CRAN

Map family


minis


map(minis, antennate)

```
antennate(minis[[1]])
antennate(minis[[2]])
antennate(minis[[3]])
antennate(minis[[4]])
antennate(minis[[5]])
```

Your turn

Look at the help for `map()`.

What other functions are documented there? How do they differ from `map()`?

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)

Your turn

Compute the mean of every column in mtcars.

Generate 10 random normals for the following means: -10, 0, 10, 100

Compute the number of unique values in each column of iris

When working with purrr

For each task, identify:

1. The output type (i.e. which map function do you need?)
2. The object that gets transformed
3. The function that transforms it

Solutions:

```
map_dbl(mtcars, mean)

mu <- c(-10, 0, 10, 100)
map(mu, rnorm, n = 10)

nunique <- function(x) length(unique(x))
map_int(iris, nunique)
# Or use an anonymous function
map_int(iris, function(x) length(unique(x)))
# Or use the formula helper
map_int(iris, ~ length(unique(.x)))
```

Which to prefer?

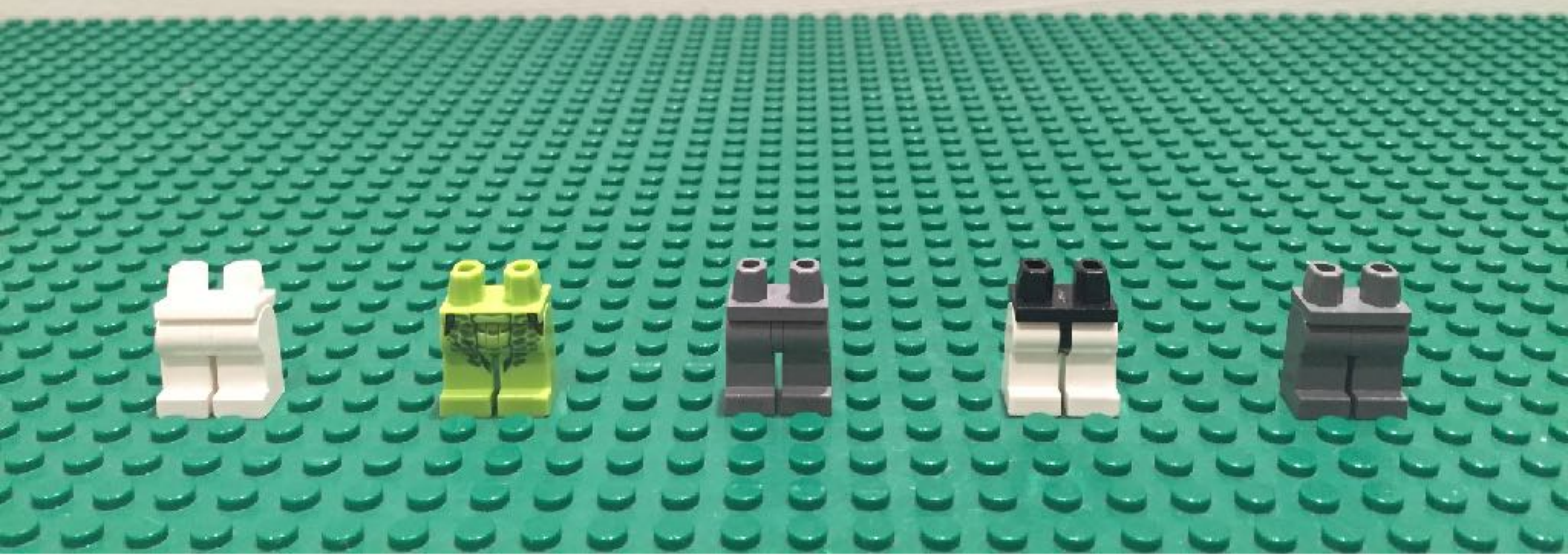
```
iris %>%
  map_int(~ length(unique(.x)))

iris %>%
  map_int(iris, . %>% unique() %>% length())

iris %>%
  map(unique) %>%
  map_int(length)
```

Two common mistakes

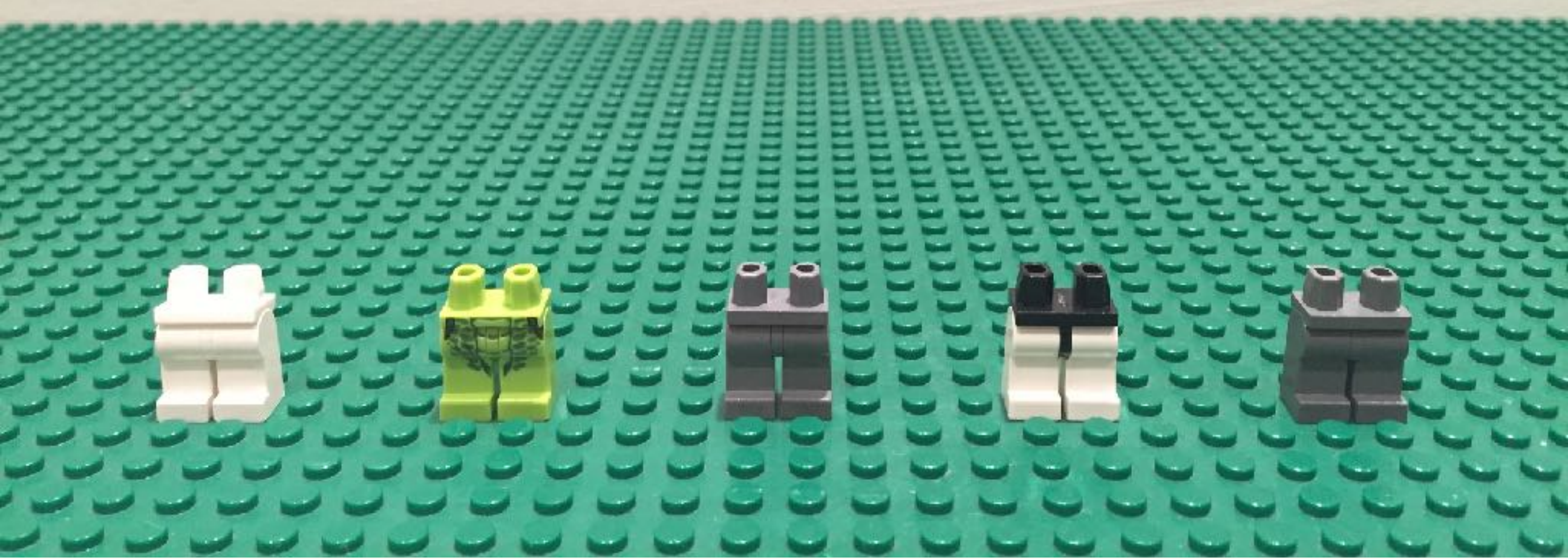
```
iris %>% map(length(unique))
iris %>% map(length(unique(.)))

# Why do they work?
length(unique)
#> [1] 1
length(unique(iris))
#> [1] 5

iris %>% map(1)
iris %>% map(5)
```

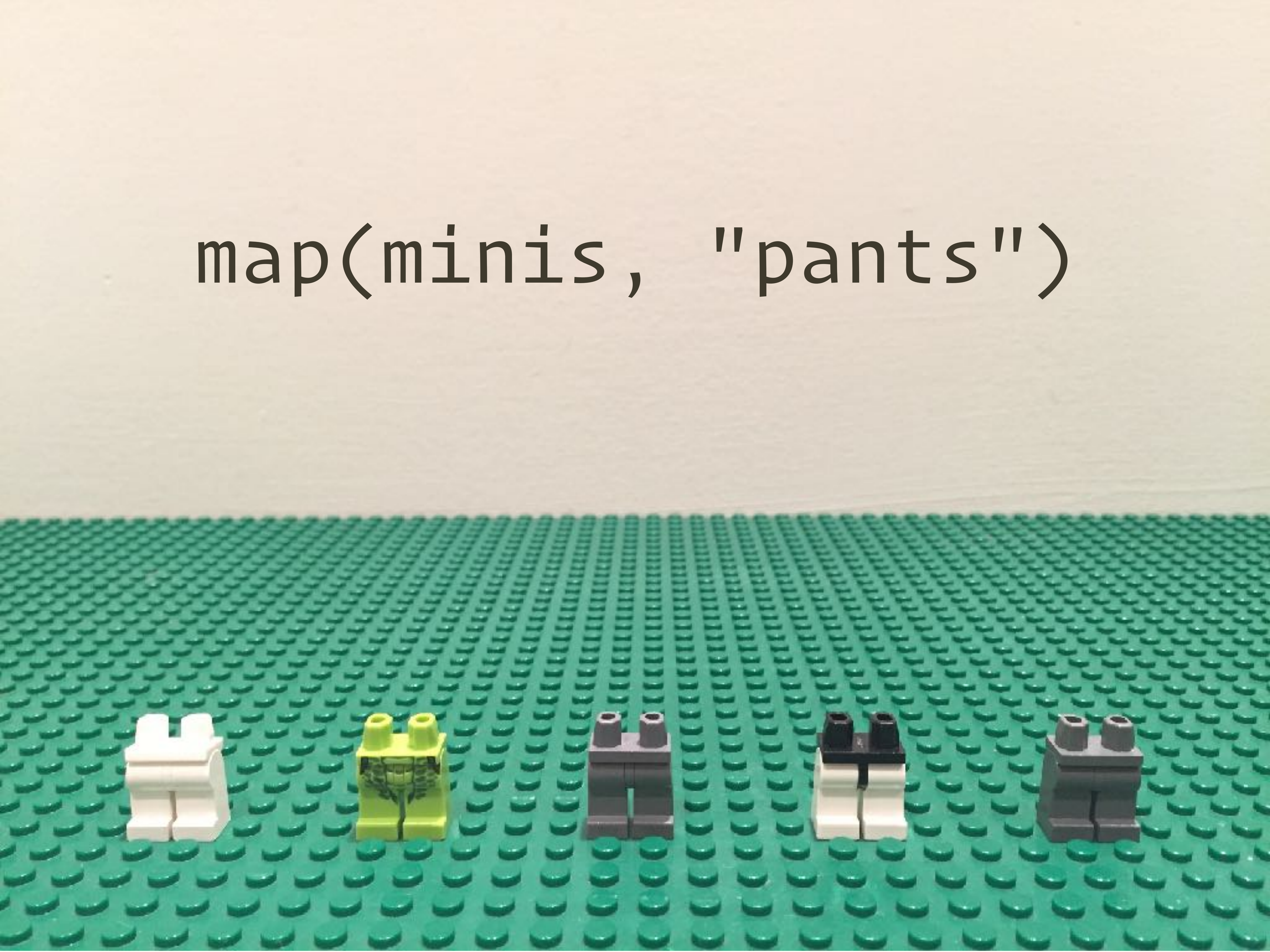

minis

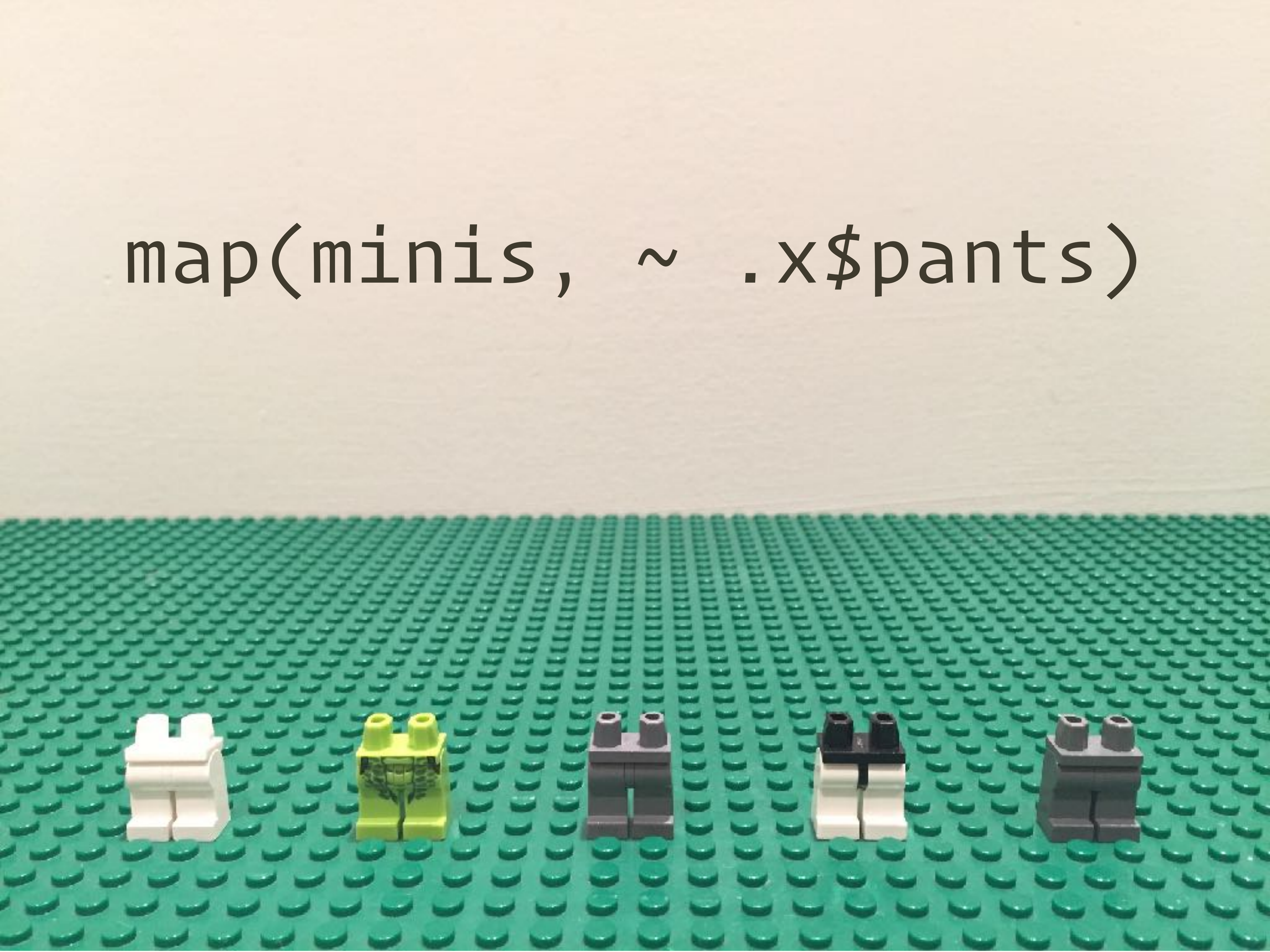
map(minis, 3)



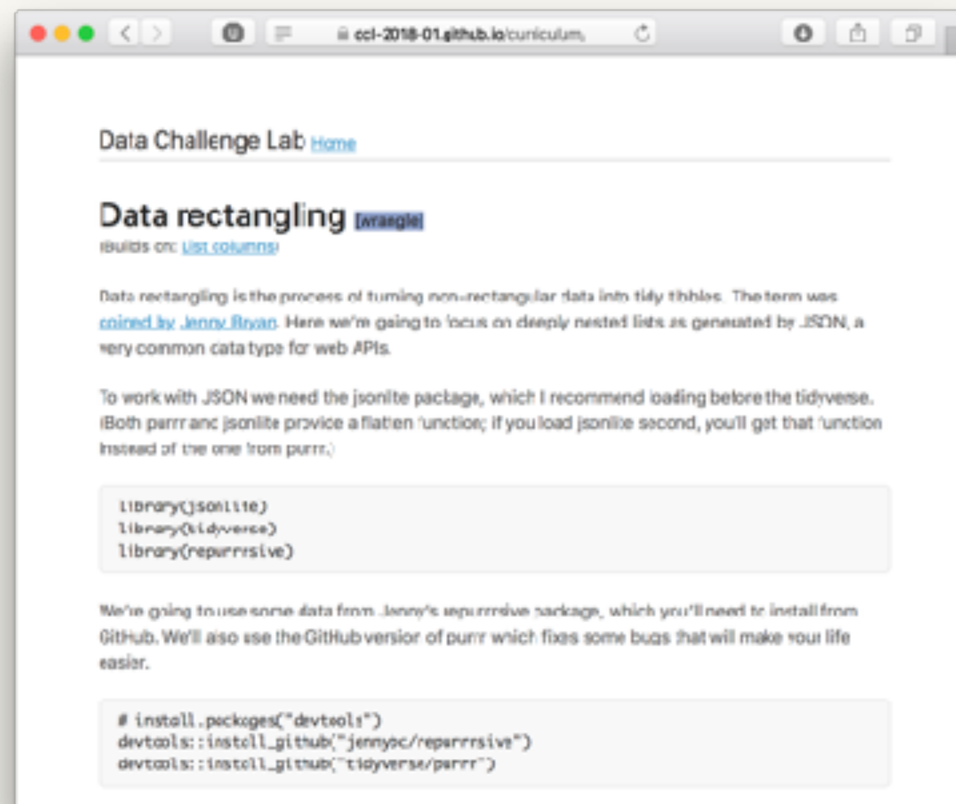
```
map(minis, ~ .x[[3]])
```




map(minis, "pants")


map(minis, ~ .x$pants)

https://speakerdeck.com/jennybc/data-rectangling



https://dcl-2018-01.github.io/curriculum/rectangling.html

purrr vs dplyr

purrr

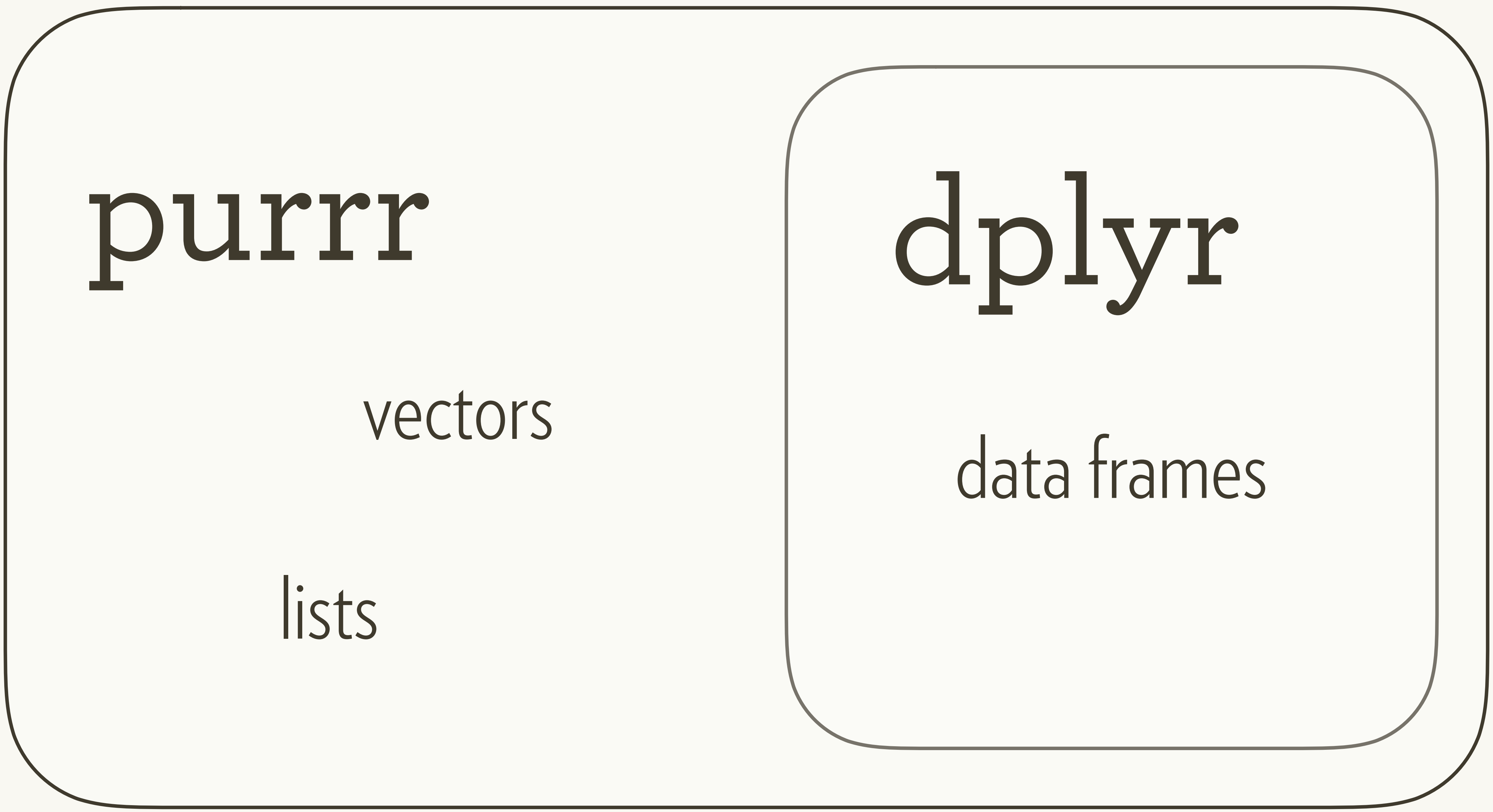
vectors

lists

dplyr

data frames

But data frames are lists



For column-wise operations you can use either purrr or dplyr

Type stability

Why is sapply challenging to program with?

```
df <- data.frame(
  a = 1L,
  b = 1.5,
  y = Sys.time(),
  z = ordered(1)
)
```

Guess the type of output

```
df[1:4] %>% sapply(class) %>% str()
df[1:2] %>% sapply(class) %>% str()
df[3:4] %>% sapply(class) %>% str()
```

Principle:
Minimise context needed to predict output type

The extreme is a type-stable function which
always returns the same type regardless of the input

The purrr alternative

```r
df <- data.frame(
  a = 1L,
  b = 1.5,
  y = Sys.time(),
  z = ordered(1)
)
```

Guess the type of output

```r
df[1:4] %>% map_chr(class) %>% str()
df[1:2] %>% map_chr(class) %>% str()
df[3:4] %>% map_chr(class) %>% str()
```

A more realistic example

```
col_means <- function(df) {
  numeric <- sapply(df, is.numeric)
  numeric_cols <- df[, numeric]

  as.data.frame(lapply(numeric_cols, mean))
}
```

What's wrong with col_means?

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])

df <- data.frame(
  x = 1:26,
  y = letters
)
col_means(df)
```

sapply and [are not type stable

```
col_means <-
```

list or logical vector

```
  numeric <- sapply(df, is.logical)
  numeric_cols <- df[, numeric]
```

vector or data frame

```
  as.data.frame(l        _cols, mean))
}
```

One possible solution

```r
col_means <- function(df) {
  numeric <- map_lgl(df, is.numeric)
  numeric_cols <- df[, numeric, drop = FALSE]

  as.data.frame(map(numeric_cols, mean))
}
```

One possible solution

always returns logical vector

```
col_means <- function(df) {
  numeric <- map_lgl(df, is.numeric)
  numeric_cols <- df[, numeric, drop = FALSE]

  as.data.frame(map(numeric_cols
}
```

always returns data frame

Can simplify further with other helpers

```
col_means <- function(df) {
  numeric_cols <- keep(df, is.numeric)
  map_dfc(numeric_cols, mean)
}
```

Is keep() type stable? It returns the output the same type as its input

Which is particularly elegant with the pipe

```
col_means <- function(df) {
  df %>%
    keep(is.numeric) %>%
    map_dfc(mean)
}
```

Type stability and stringr

```
# Instead of suffixes
str_replace()
str_replace_all()

# could use an argument
str_replace(n = 1)
str_replace(n = Inf)

# which generalises better
str_replace(n = 2)
str_replace(n = -1)
```

But that would violate type stability

```
strings <- c("x y", "x y x")
str_locate(strings, "x", n = 1)
#>      start end
#> [1,]     1   1
#> [2,]     1   1

str_locate(strings, "x", n = Inf)
#> [[1]]
#>      start end
#> [1,]     1   1
#>
#> [[2]]
#>      start end
#> [1,]     1   1
#> [2,]     5   5
```

What type does this code return?

```
str_locate(strings, "x", n = x)
```

Handling errors

What happens when there is an error?

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

Principle:

Turn side-effects into data

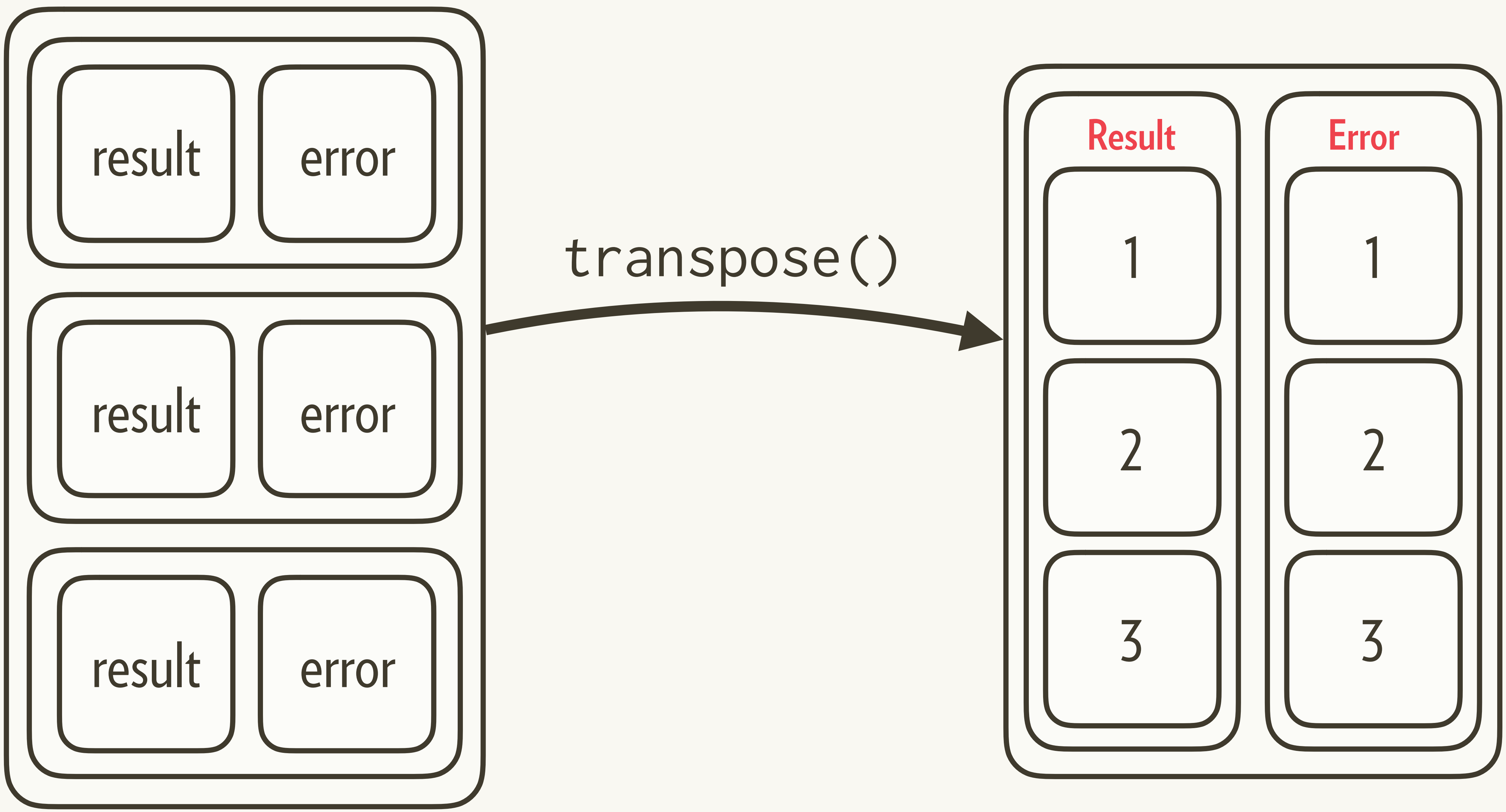
What does safely() do?

```
# safely() modifies a function so it never fails
input <- list(1:10, sqrt(4), 5, "n")
map(input, safely(log))

# What does it return when the function succeeds?
# What does it return when the function fails?
```

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

Apply transpose() to the previous result then:

1. List failed urls

2. 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]
```

Parallel maps

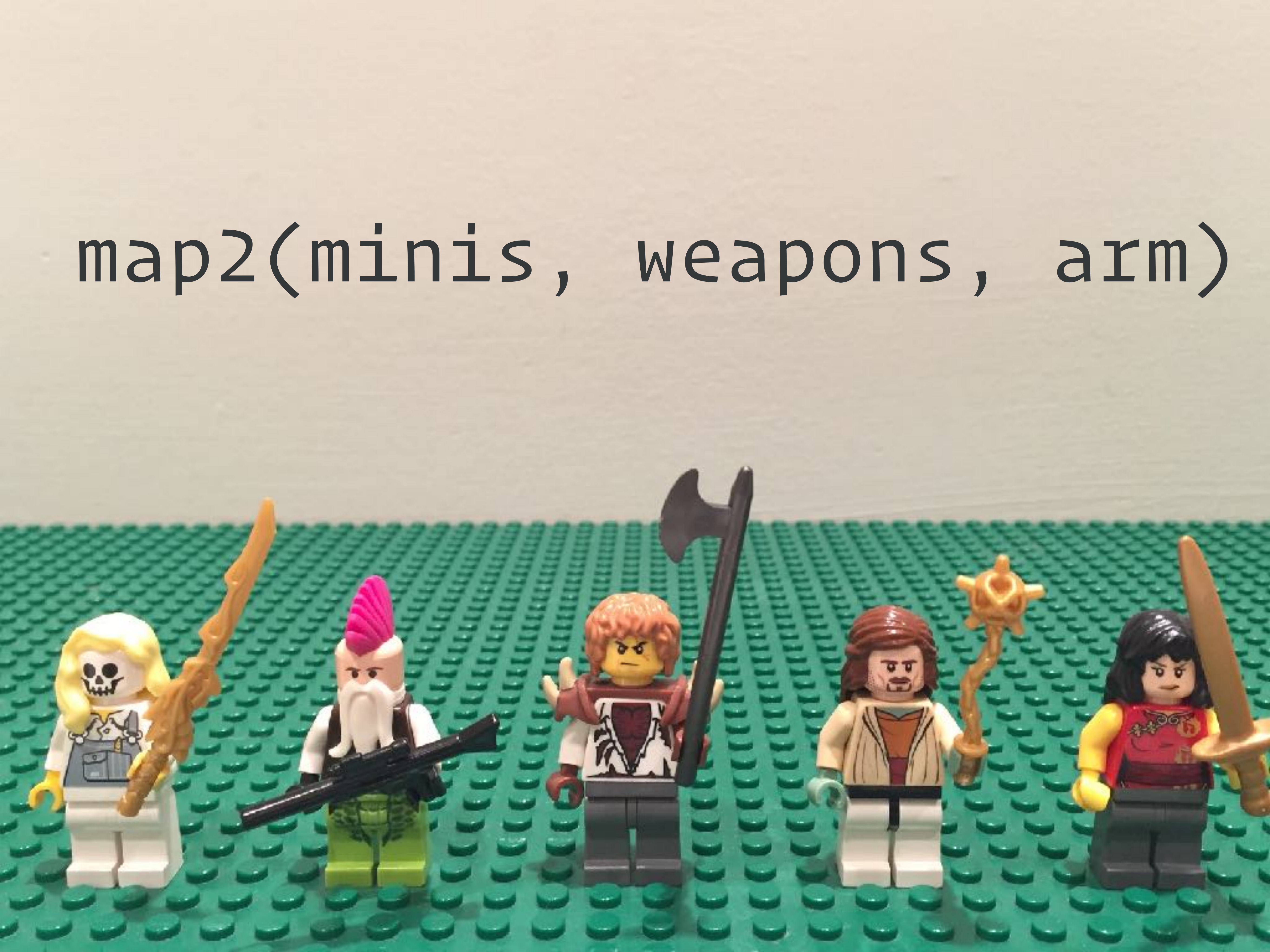
(Not parallel programming, sorry!)


hair
minis

```
enhair(minis[[1]], hair[[1]])
enhair(minis[[2]], hair[[2]])
enhair(minis[[3]], hair[[3]])
enhair(minis[[4]], hair[[4]])
enhair(minis[[5]], hair[[5]])
```

```
map2(minis, hair, enhair)
```


weapons
minis


map2(minis, weapons, arm)

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 col matrix
# or two vectors
```

What if we have multiple locations?

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

pos[[1]]

# How do we generalise?
str_sub(sentences[[1]], pos[[1]])
str_sub(sentences[[2]], pos[[2]])
str_sub(sentences[[3]], pos[[3]])
```

1	`map()`
2	`map2()`
1 + index	`imap()`
3+	`pmap()`
fun	`invoke_map()`

Isolate side effects

Principle:

It’s easier to understand a function when it has either a **side effect** or a **return value**

Your turn

Any action other than returning a value is a **side-effect**.
What are some common side-effects in base R?

Some important side-effects

```
plot()
write.csv()

print()
message() / warning() / stop()

library(dplyr)
x <- 1
setClass() etc
options()
par()
setwd()
```

A few functions legitimately need to do both

```r
# One exception are random number generators

.Random.seed[2]
#> [1] 624

runif(5)
#> [1] 0.0808 0.8343 0.6008 0.1572 0.0074

.Random.seed[2]
#> [1] 5
```

summary() is a interesting mix

```
x <- runif(100)
summary(x)

# But actually two parts
y <- summary(x)
str(y)
print(y)

# This is a very useful technique
# We'll come back to this in OO programming
```

Same idea in ggplot2

```
library(ggplot2)
p <- ggplot(mpg, aes(mpg, wt)) +
  geom_point()
str(p)
print(p)

# This works because of implicit printing:
# results of most R expressions are
# automatically printed. Makes it
# possible to return value and have one
# side effect when used interactively
```

Principle:

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

walk() returns its output **invisibly**

```
species <- split(iris, iris$Species)
file_name <- paste0(names(species), ".csv")

walk2(species, file_name, readr::write_csv)
```

Your turn

Create a col_write(df, path) function that writes out each column into a separate file named colname.txt, which one value on each line.

The package includes a unit test that you can use to check your work.