Functional programming

September 2016

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
```

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

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

- 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

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)
}
```

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

```
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[] <- map(df, fix_missing)

# cf
df <- map(df, fix_missing)
```

And provide useful tools for generalisation

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

numeric <- map_lgl(df, is_numeric)
df[numeric] <- map(df[numeric], fix_missing)

# OR
df <- map_if(df, is_numeric, fix_missing)
```

Warmups

Your turn

What's the difference between [, [[and $?

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

If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6.

http://twitter.com/#!/RLangTip/status/118339256388304896

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

Beware of partial matching

```
x <- list(abc = 1)
x[["abc"]]
x$abc

x$a

# One solution:
options(
  warnPartialMatchArgs = TRUE,
  warnPartialMatchDollar = TRUE,
  warnPartialMatchAttr = TRUE
)
```

What does this code do?

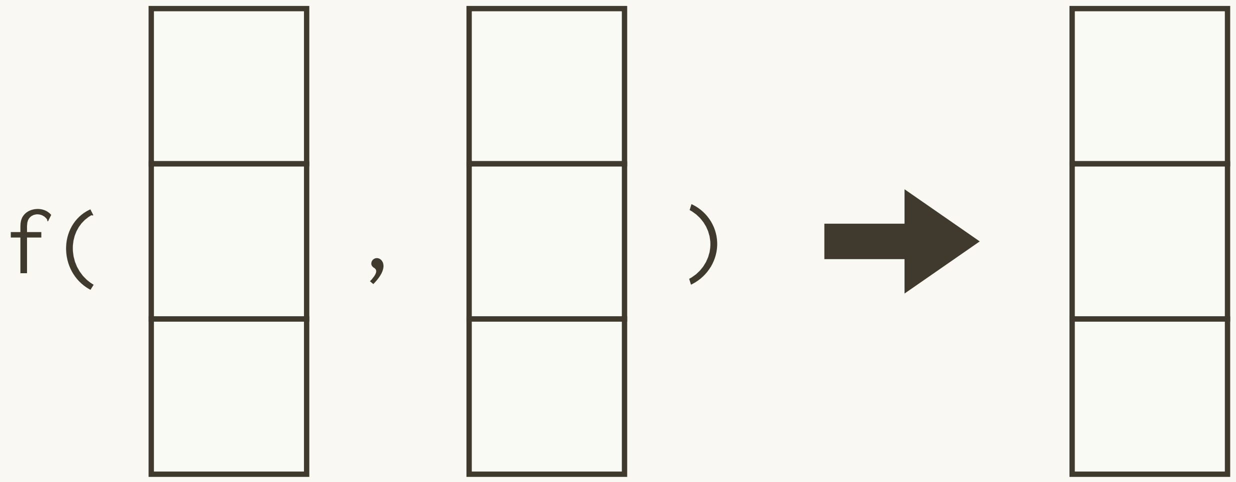
```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
for(var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

How could you reduce duplication here?

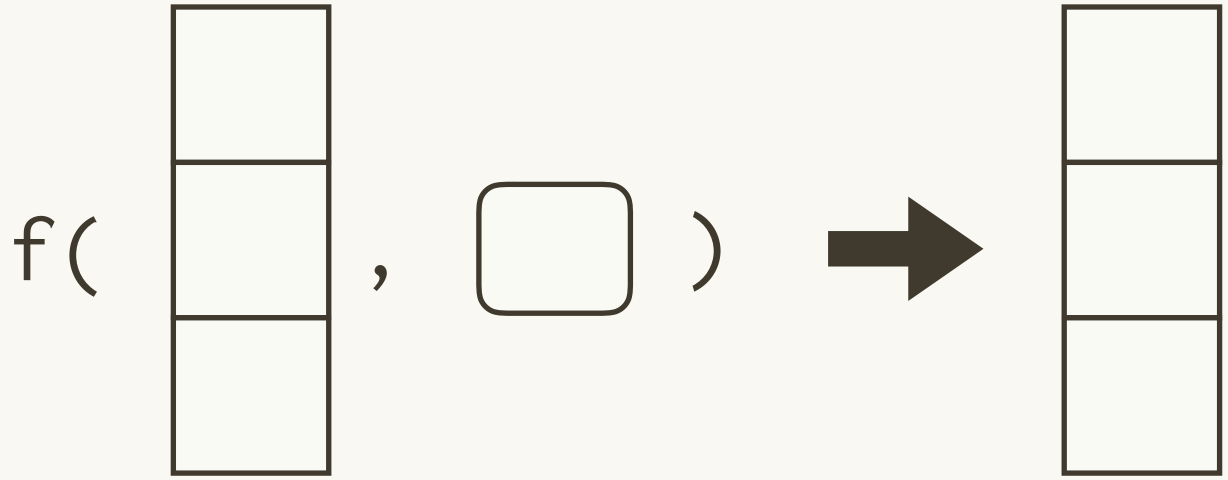
```
f1 <- function(x) x + 1
f2 <- function(x) x + 2
f3 <- function(x) x + 3
f4 <- function(x) x + 4
```

Functionals

What is a functional?



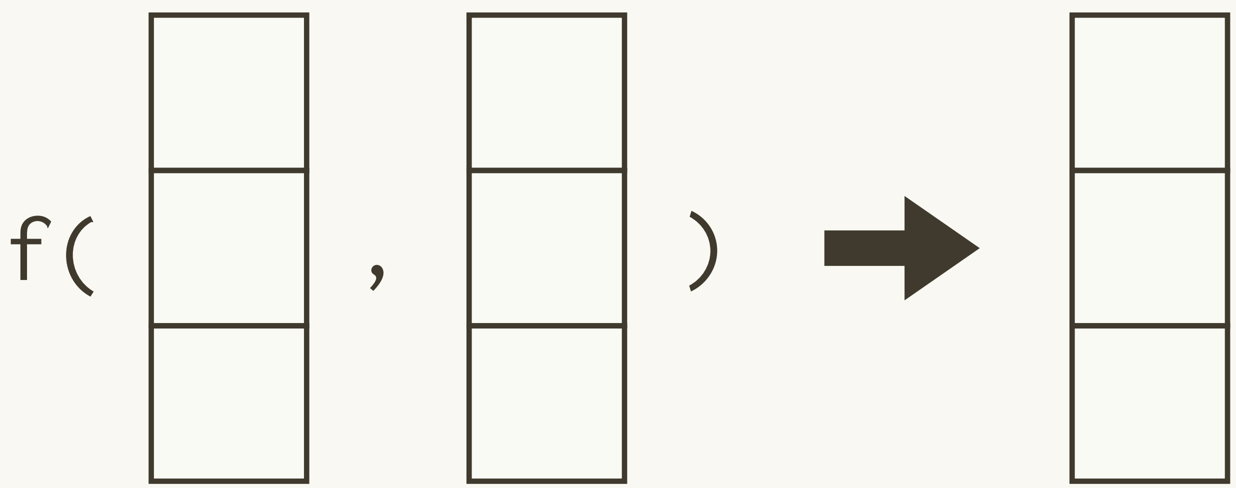
Most functions take vectors as inputs and return vectors as output



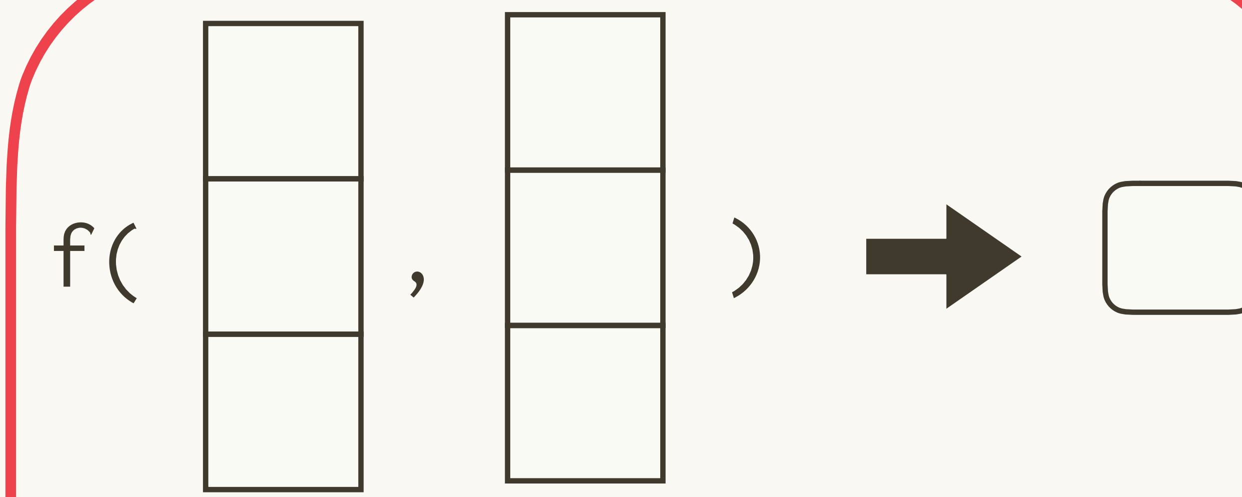
Functionals take a function as an argument, and return a vector

What is a functional?

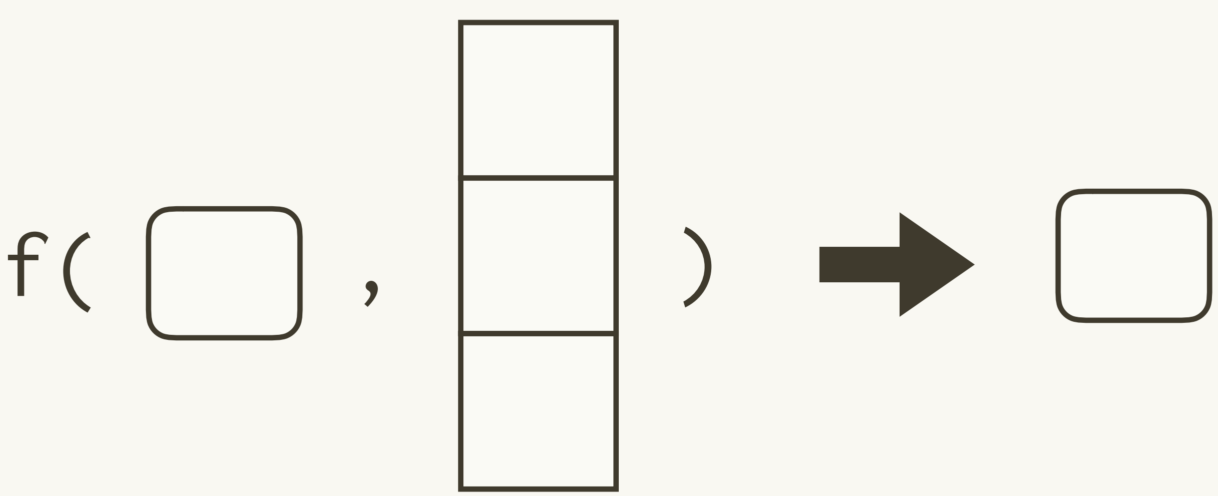
Higher order functions



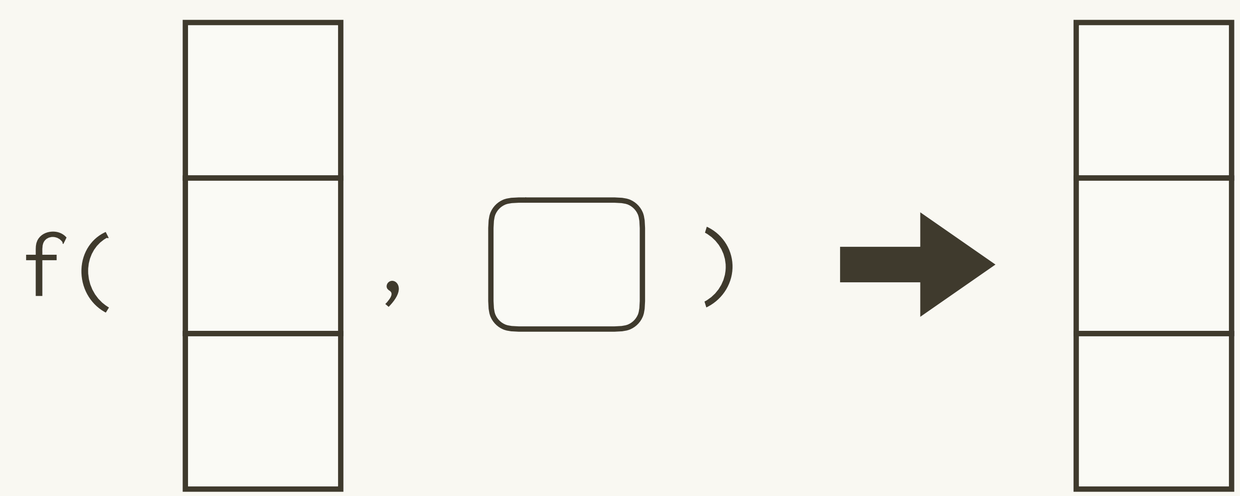
Most functions take vectors as inputs and return vectors as output



Function factories take vectors as input and return a function



Function operators take functions as input and return a function



Functionals take a function as an argument, and return a vector

Let's start with some stereotypical input

```
library(purrr)
set.seed(1014)

l <- rerun(20, runif(sample(1:10, 1)))
str(l)
l
```

How can we extract the length of each element?

```
lengths <- numeric(length(l))
for (i in seq_along(l)) {
  lengths[[i]] <- length(l[[i]])
}
lengths
```

All for loops follow this basic pattern

Preallocating space for output saves a lot of time

```
lengths <- numeric(length(l))
for (i in seq_along(l)) {
  lengths[[i]] <- length(l[[i]])
}
lengths
```

Safe shortcut for `1:length(l)`

How could you compute the mean of each element?

```
compute_length <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- length(x[[i]])
  }
  out
}
```

What about the median?

```
compute_mean <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- mean(x[[i]])
  }
  out
}
```

What about the standard deviation?

```
compute_median <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- median(x[[i]])
  }
  out
}
```

How could you reduce the duplication?

```
compute_length <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- length(x[[i]])
  }
  out
}
compute_mean <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- mean(x[[i]])
  }
  out
}
compute_median <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- median(x[[i]])
  }
  out
}
compute_sd <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- sd(x[[i]])
  }
  out
}
```

A brief diversion: Which is “better”?

```r
f1 <- function(x) x + 1
f2 <- function(x) x + 2
f3 <- function(x) x + 3
f4 <- function(x) x + 4

# Or:
f <- function(x, y) x + y
```

Your turn

How could you reduce the duplication between `compute_length()` and `compute_mean()` and `compute_median()`?

Use template on next slide

Use this template:

```
compute <- function(?, ?) {
  ...
}

compute(l, length)
compute(l, mean)
compute(l, median)
```

Stuck? Try hint on next slide

How are these the same?

```
compute_length <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- length(x[[i]])
  }
  out
}
compute_mean <- function(x) {
  out <- numeric(length(x))
  for (i in seq_along(x)) {
    out[[i]] <- mean(x[[i]])
  }
  out
}
```

```
f1 <- function(x) x + 1
f2 <- function(x) x + 2

# versus
f <- function(x, y) x + y
```

No peeking until you've made an attempt!

Functions can be arguments!

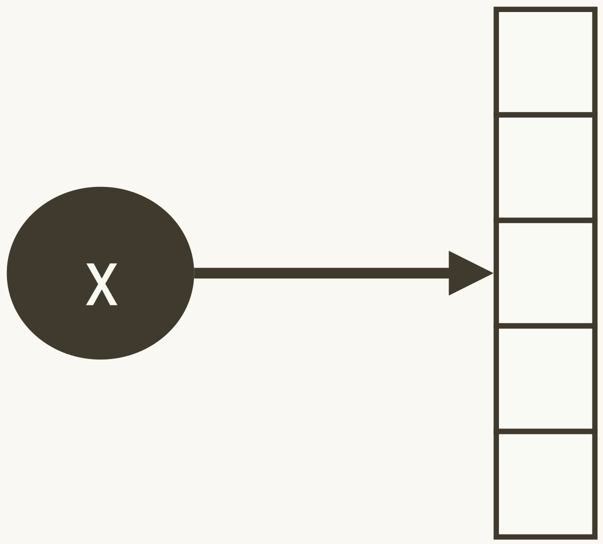
```
compute <- function(x, f) {
  out <- numeric(length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]])
  }
  out
}

compute(l, length)
compute(l, mean)
compute(l, median)
```

Functions can be arguments!

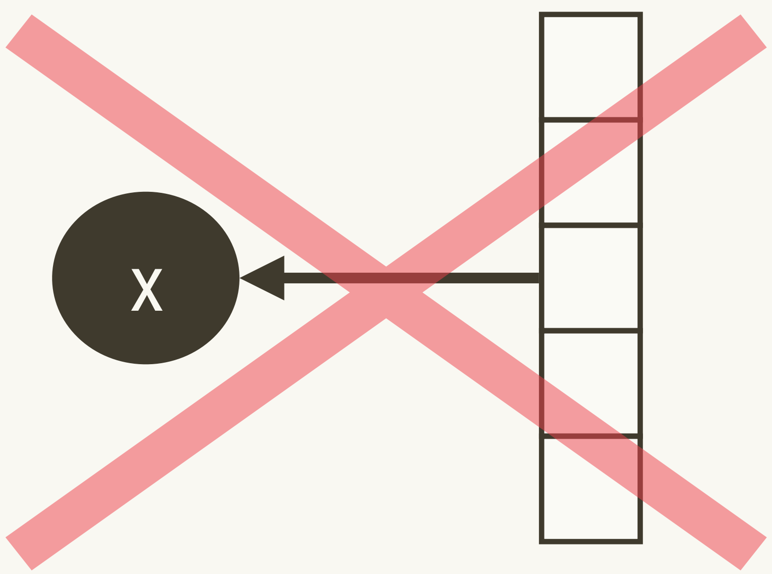
```
compute <- function(x, f) {
  out <- numeric(length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]])
  }
  out
}

compute(l, length)
compute(l, mean)
compute(l, median)
```

```
function (x)  .Primitive("length")
```

Remember!



A name "has" an object



An object doesn't have a name

But wait!

```
lapply(l, length)
lapply(l, mean)
lapply(l, median)

# We've just invented something very
# close to lapply :)

# Three differences:
# * lapply() gives a list instead of numeric vector
# * lapply() uses some C tricks to be faster
# * lapply() passes ... on to f
```

```r
lapply <- function(x, f, ...) {
  out <- vector("list", length(x))
  for(i in seq_along(x)) {
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

Instead of base functions we'll focus on purrr

```
library(purrr)

map_dbl(l, length)
map_dbl(l, mean)
map_dbl(l, median)

# Why?
# * More consistent
# * Handy helpers
# * More tools
```

Can’t remember how many Rs?

```
library(purrr)
library(dplyr)
library(ggvis)
library(tidyr)
library(readr)
```

Map output

Your turn

Look at the help for `map_dbl()`.

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

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_df()`	Data frame
`walk()`	Nothing

Your turn

Compute the mean of every column in mtcars.

Generate 10 random normals for each of $\mu = -10, 0, 10, 100$

Compute the number of unique values in each column of iris.

When working with purrr

For each task, identity:

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(.)))
```

Why sapply is dangerous

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

Guess the type of output

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

The purrr alternative

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

Guess the type of output

```
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]

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

What's wrong with that function?

```
col_means(mtcars)
col_means(mtcars[, 0])
col_means(mtcars[0, ])
col_means(mtcars[, "mpg", drop = F])
col_means(1:10)
col_means(as.matrix(mtcars))

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

sapply and [are not type stable

list or logical vector

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

  data.frame(lapply mean))
}
```

vector or data frame

Your turn

Rewrite col_means avoiding type-unstable functions.

Can you generalise col_means to col_sum with an arbitrary summary function?

One possible solution

```
col_means <- function(df) {
  stopifnot(is.data.frame(df))

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

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

One possible solution

```
col_means <- function(df) {
  stopifnot(is.(
```

always returns logical vector

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

always returns data frame

```
  data.frame(map(numeric_cols, m
}
```

Keep helps with this common pattern

```
col_means <- function(df) {
  stopifnot(is.data.frame(df))

  numeric_cols <- keep(df, is.numeric)
  as.data.frame(map(numeric_cols, mean))
}
```

If you like piping:

```
col_means <- function(df) {
  stopifnot(is.data.frame(df))

  df %>%
    keep(is.numeric) %>%
    map(mean) %>%
    as.data.frame()
}
```

You could also consider returning a numeric vector

```
col_means <- function(df) {
  stopifnot(is.data.frame(df))

  df %>%
    keep(is.numeric) %>%
    map_dbl(mean)
}
```

Map helpers

map() is the most general tool because it makes lists

```
# Lets start by making a list of data frames
by_cyl <- split(mtcars, mtcars$cyl)

# What does that list look like?
str(by_cyl)
by_cyl[1]
by_cyl[[1]]
```

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

How could we fit the same model to each data frame?

```
# Longest
model <- function(df) lm(mpg ~ wt, data = df)
map(by_cyl, model)

# Shorter
map(by_cyl, function(df) lm(mpg ~ wt, data = df))

# Shortest (only for purrr)
map(by_cyl, ~ lm(mpg ~ wt, data = .))
```

A pronoun, like "it"

Your turn

Compute `summary()` for each model. What component gives R^2?

Extract the coefficients for each model.

```r
models <- map(by_cyl, ~ lm(mpg ~ wt, data = .))

map(models, summary)
str(summary(models[[1]]))

map(models, coef)
# Easier to work with
map(models, broom::tidy)
```

Three ways to extract R2

```
summaries <- map(models, summary)

map_dbl(summaries, function(x) x$r.squared)
map_dbl(summaries, ~ .$r.squared)
map_dbl(summaries, "r.squared")

# Not a good idea here!
map_dbl(summaries, 8)
```

Handling errors

What happens when there is an error?

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

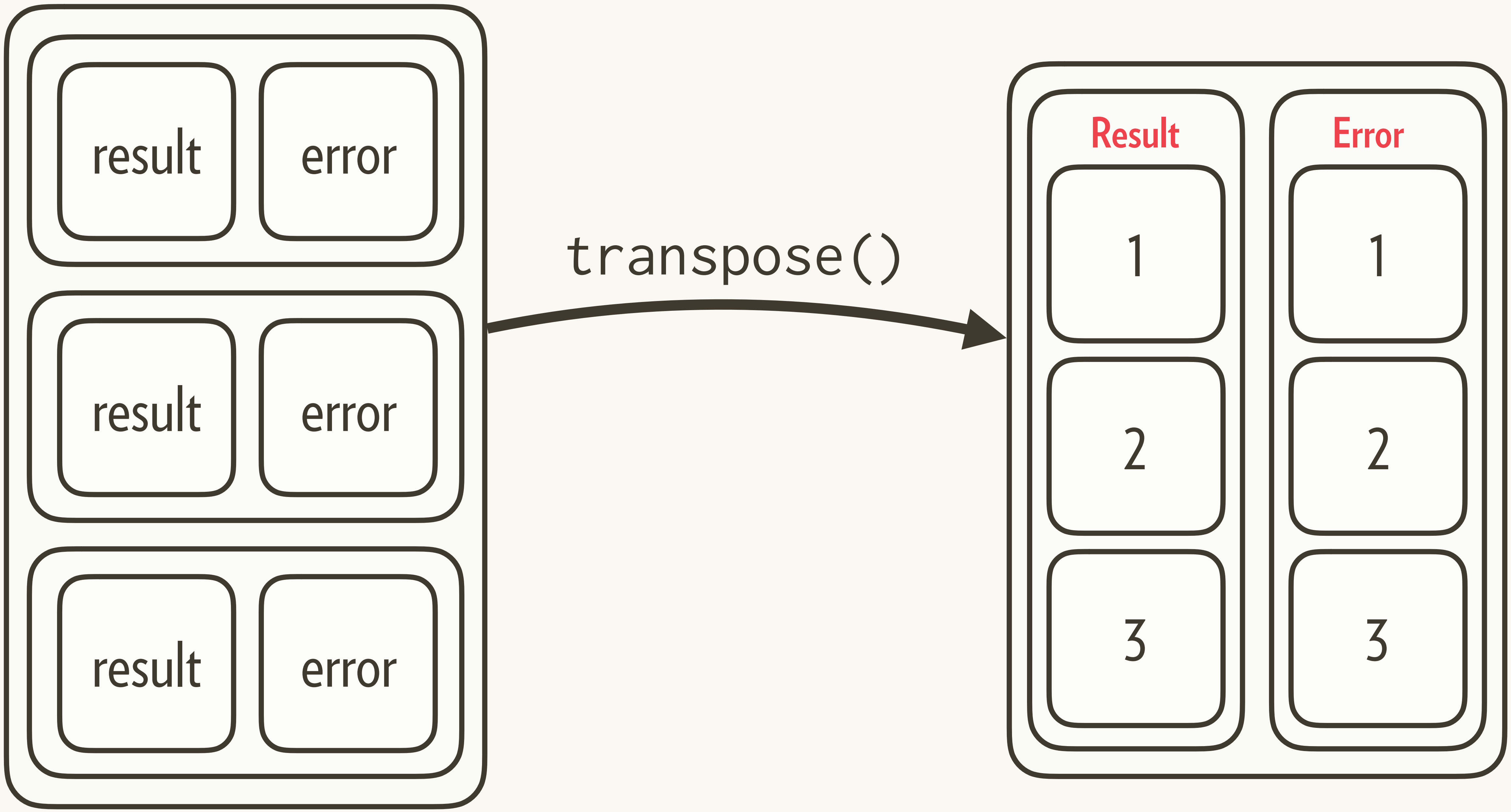
What does safely() do?

```
# safely() modifies a function so it never fails
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(
  "http://google.com",
  "https://en.wikipedia.org",
  "asdfasdasdkfjlda"
)

# Fails
contents <- map(urls, readLines)

# Always succeeds
contents <- urls %>% map(safely(readLines))
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!)

What if you want to vary more than one input?

```
# Imagine we want to simulate some random
# normals, varying the mean:
means <- c(5, 10, -3)
map(means, ~ rnorm(10, mean = .))

# Alternatively, we could vary the sd:
sds <- c(1, 2, 3)
map(sds, ~ rnorm(10, sd = .))

# What if we want to vary both?
```

map2() works like map() but varies two arguments

```r
mean <- c(5, 10, -3)
sd <- c(1, 2, 3)

map2(mean, sd, ~ rnorm(10, mean = .x, sd = .y))
#> list(
#>   rnorm(10, mean = 5, sd = 1),
#>   rnorm(10, mean = 10, sd = 2),
#>   rnorm(10, mean = -3, sd = 3)
#> )

# Or
map2(mean, sd, rnorm, n = 10)

# The pipe doesn't feel natural with two
# equally important inputs
```

Instead of map3(), map4(), etc we have pmap()

```
args <- list(
  mean = c(5, 10, -3),
  sd = c(1, 2, 3),
  n =  c(3, 5, 7)
)

pmap(args, rnorm)
```

All the arguments must have the same length…

```
args <- tibble::frame_data(
  ~mean, ~sd, ~n,
      5,   1,  3,
     10,   3,  2,
     -3,   5,  7
)

args %>% pmap(rnorm)

# Next challenge: what if the function also varied?
```

invoke_map() also lets you vary the function

```r
f <- list(runif, rnorm, rpois)
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)

invoke_map(f, param, n = 5) %>% str()
```

Cupcakes

	Flour	Baking powder	Sugar	Butter	Egg	Extra
Vanilla	120	1.5	140	40	1	0.25t vanilla
Chocolate	100	1.5	140	40	1	20g cocoa • 0.25t vanilla
Lemon	120	1.5	140	40	1	2T lemon zest
Red velvet	150	0	150	60	1	10g cocoa • 20ml red colouring • 1.5t vinegar • 0.5 t baking soda

5. Store as data

Your turn

The "shape" of the arguments list is different between `pmap()` and `invoke_map()`.

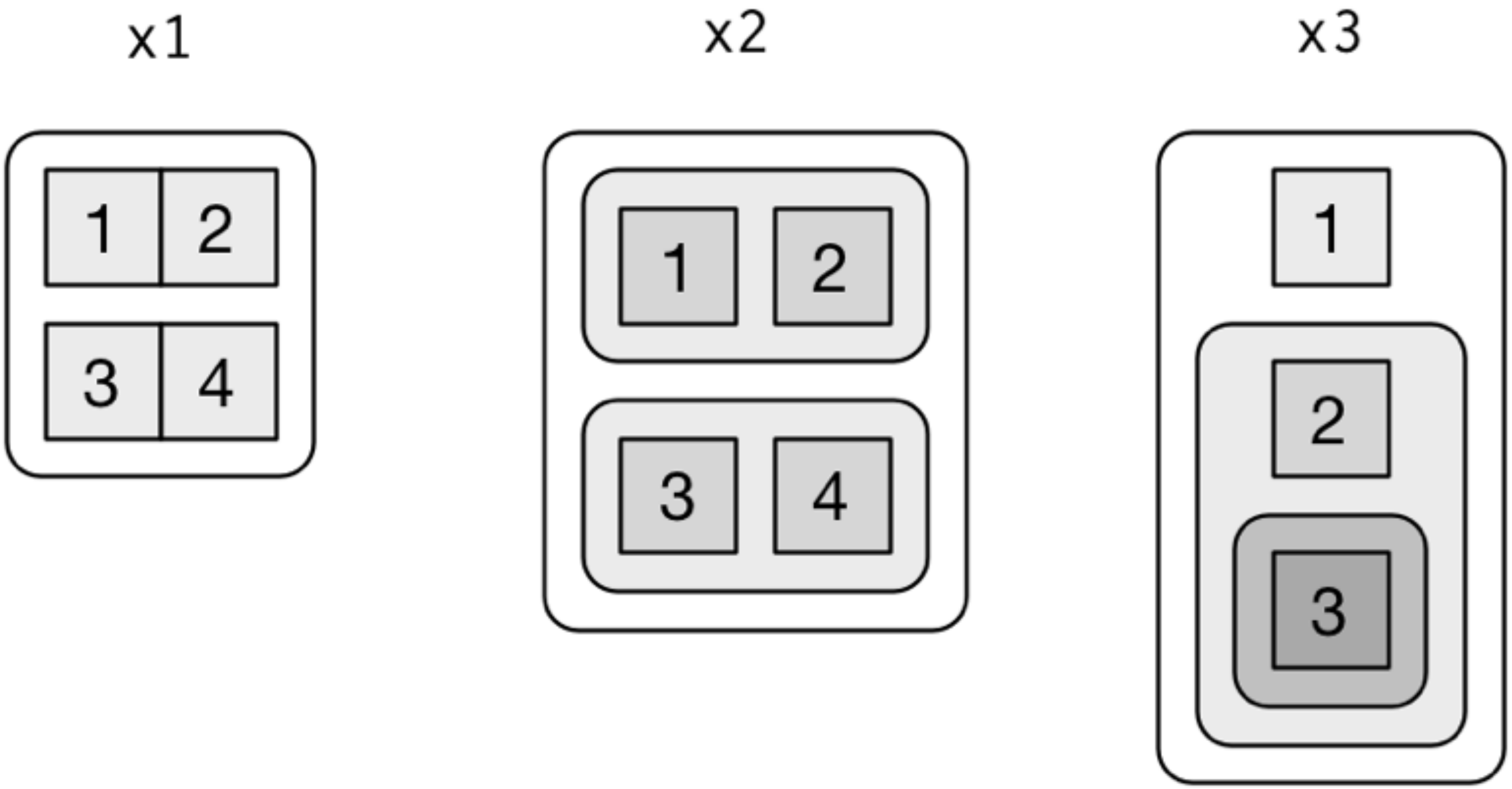
How would you characterise the difference? Why can't they be consistent?

```r
# pmap()
list(
  mean = list(5, 10, -3),
  sd = list(1, 2, 3),
  n =  list(3, 5, 7)
)
x$sd[[1]]

# invoke_map()
list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
x[[2]]$sd
```

Learning more

R for data science: http://r4ds.had.co.nz/iteration.html

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



http://adv-r.had.co.nz/Functions.html

http://adv-r.had.co.nz/Functional-programming.html