Functional programming

April 2019

Hadley Wickham
@hadleywickham

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

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

120g flour

140g sugar

1.5 t baking powder

40g unsalted butter

120ml milk

1 egg

0.25 t pure vanilla extract

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

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

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?

```r
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 weights equally

```
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)
```

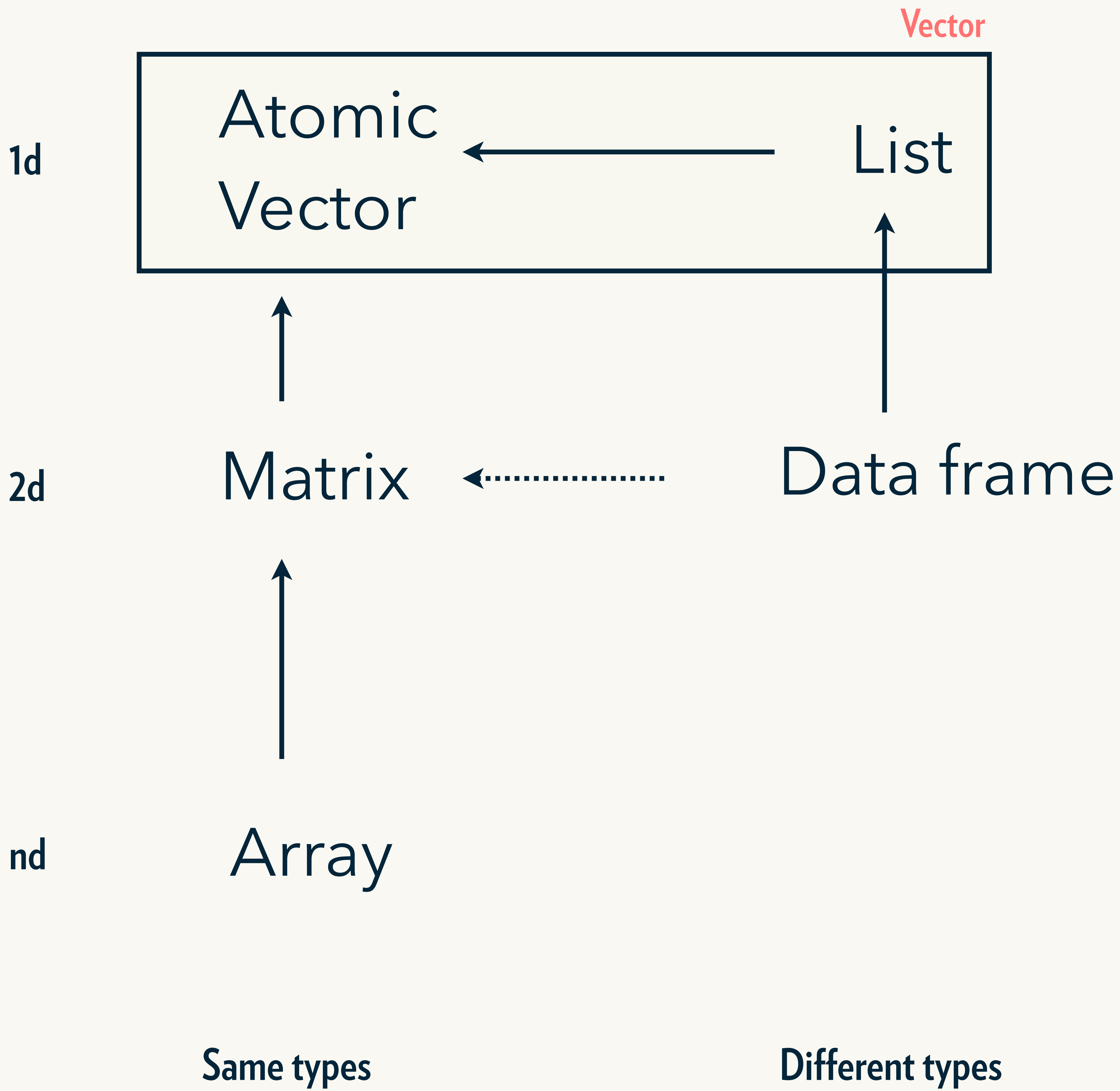
Warmups

Your turn

How is a list different from an atomic vector?

How is a data frame different from a list?

How do you examine the structure of an object?



https://adv-r.hadley.nz/vectors-chap.html

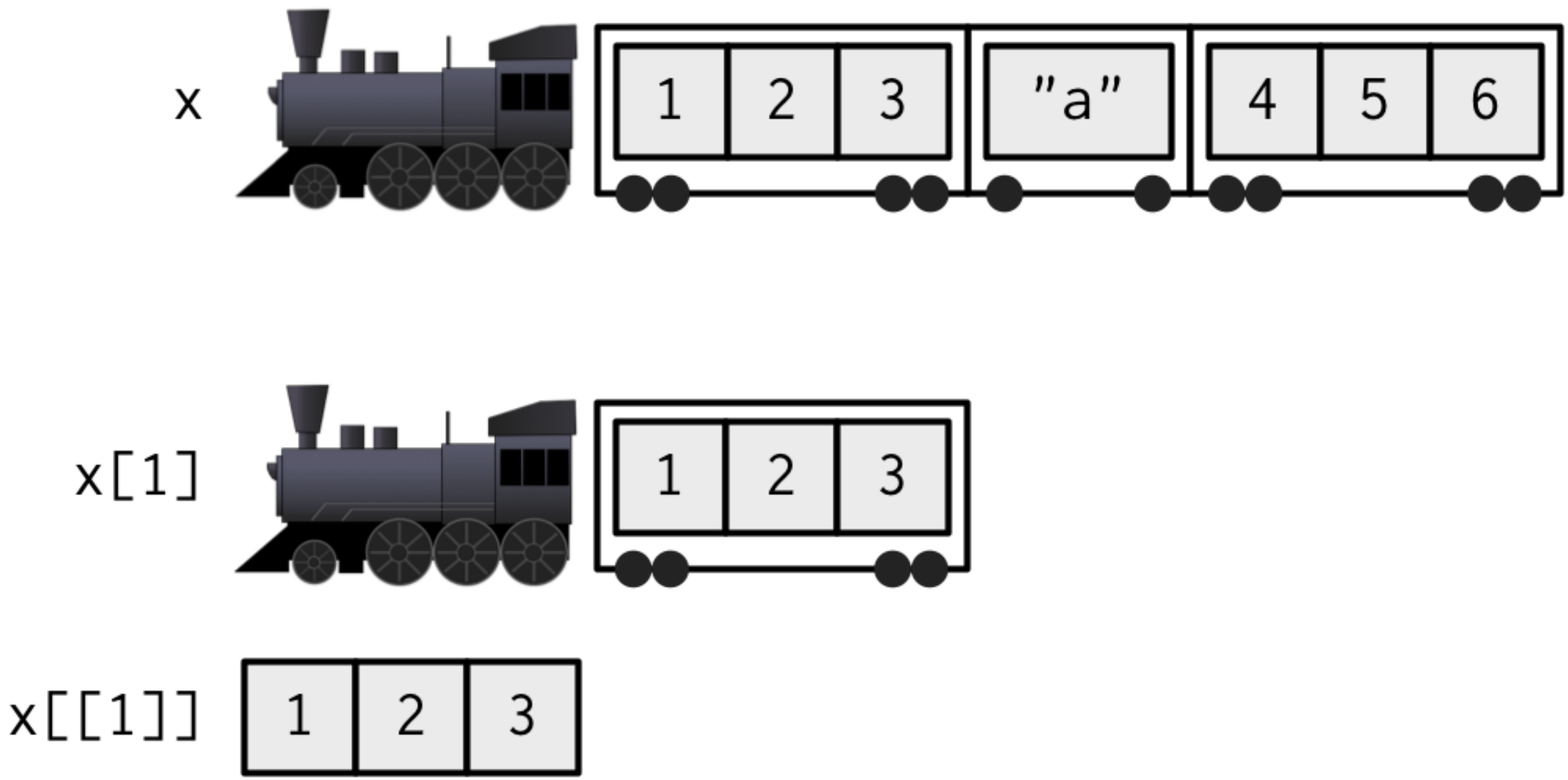
```
str()
View()
```

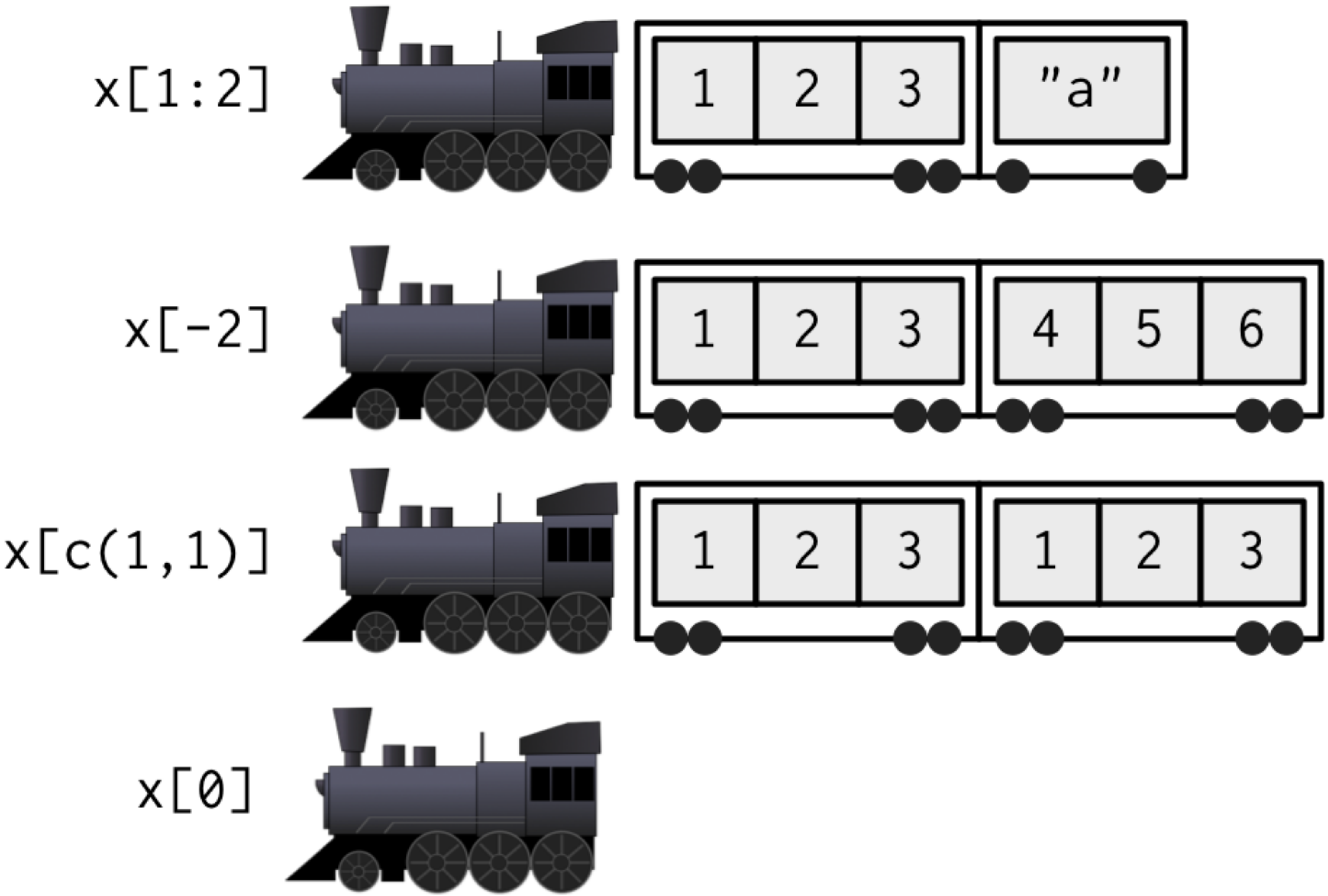
Your turn

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

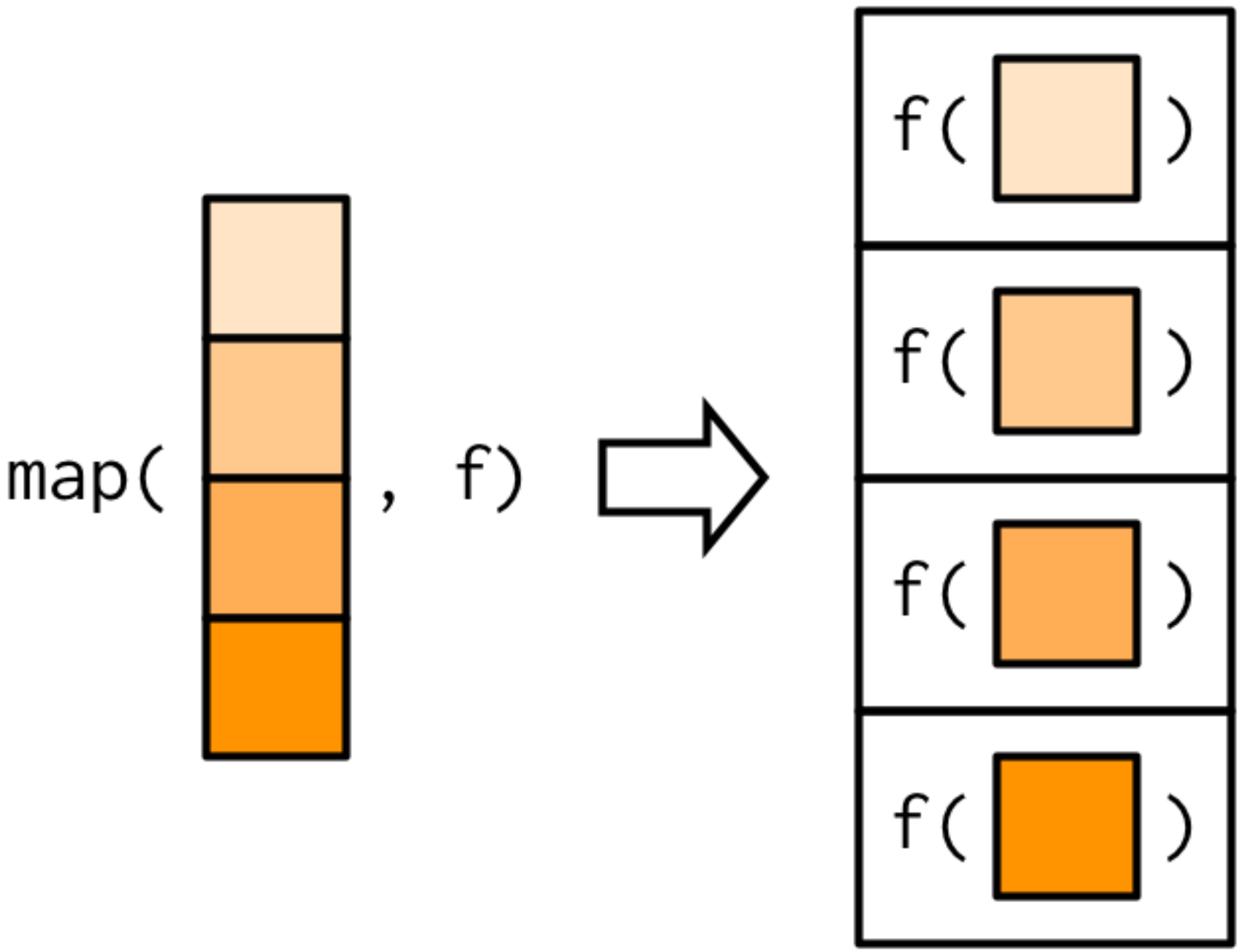

x
1
2
3
"a"
4
5
6
x[1]
1
2
3
x[[1]]
1
2
3


x[1:2]
1
2
3
"a"
x[-2]
1
2
3
4
5
6
x[c(1,1)]
1
2
3
1
2
3
x[0]

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]])
}
```

Map family



Equivalent to lapply()

Map strategy

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

Find first element of compound string

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

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

# For each element of strings_split
# 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 <- strings_split[[1]]
.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(strings_split, ~ .x[[1]])
```

For each element of strings_split,

extract its first element

Map strategy

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

What do these functions return?

Function	Output
map_lgl()	
map_int()	
map_dbl()	
map_chr()	
map()	List
map_dfc()	
map_dfr()	

Guaranteed type, or an error

```
map(strings_split, ~ .x[[1]]) %>% str()
#> List of 4
#>  $ : chr "a"
#>  $ : chr "a"
#>  $ : chr "d"
#>  $ : chr "b"

map_chr(strings_split, ~ .x[[1]])
#> [1] "a" "a" "d" "b"

map_dbl(strings_split, ~ .x[[1]])
#> Error: Can't coerce element 1 from
#> a character to a doubl>e
```

Map strategy

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

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

Compute the mean of every column in mtcars

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

# Generalise
map_dbl(mtcars, ~ mean(.x))

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

Generate 10 random normals

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

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

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

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

Compute the number of unique values in each column

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

# Generalise
map_int(iris, ~ length(unique(.x)))

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

Why not base R?

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()`	

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()`

Compared to purrr, base R functions:

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

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

Require functions (no ~, or extract helpers)

Lack paired maps (no `map2()`)

Lack side-effect form (no `walk()`)

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

Compared to purrr, base R functions:

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

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

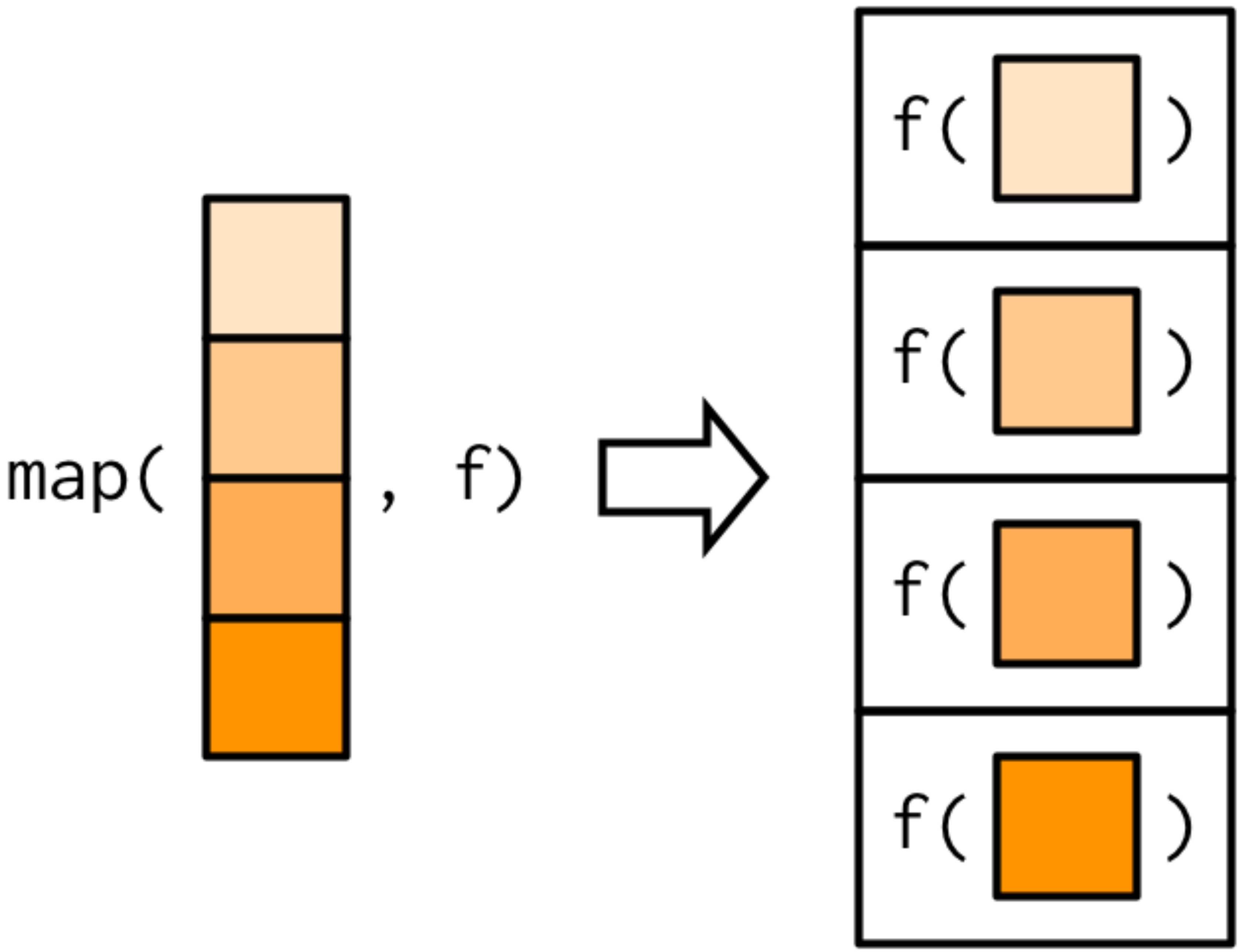
Require functions (no ~, or extract helpers)

Lack paired maps (no `map2()`)

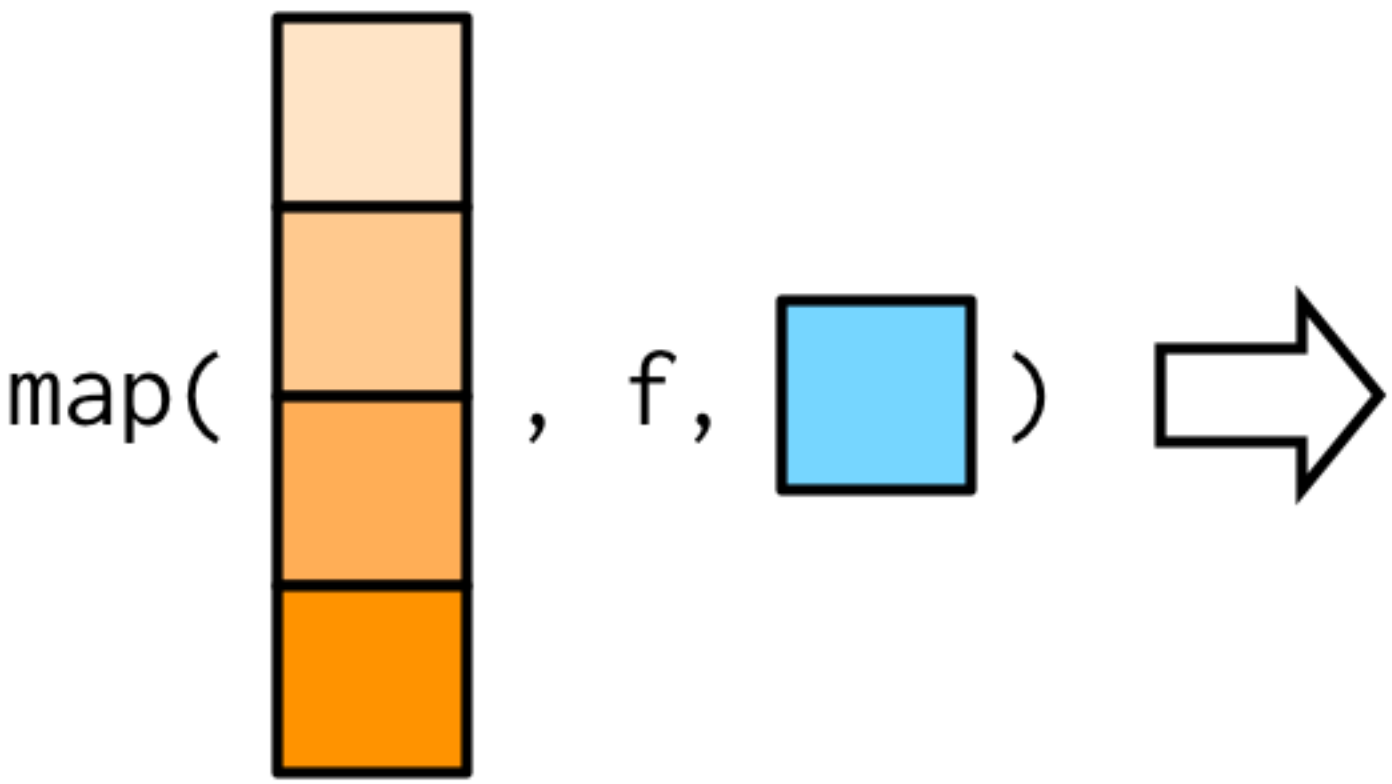
Lack side-effect form (no `walk()`)

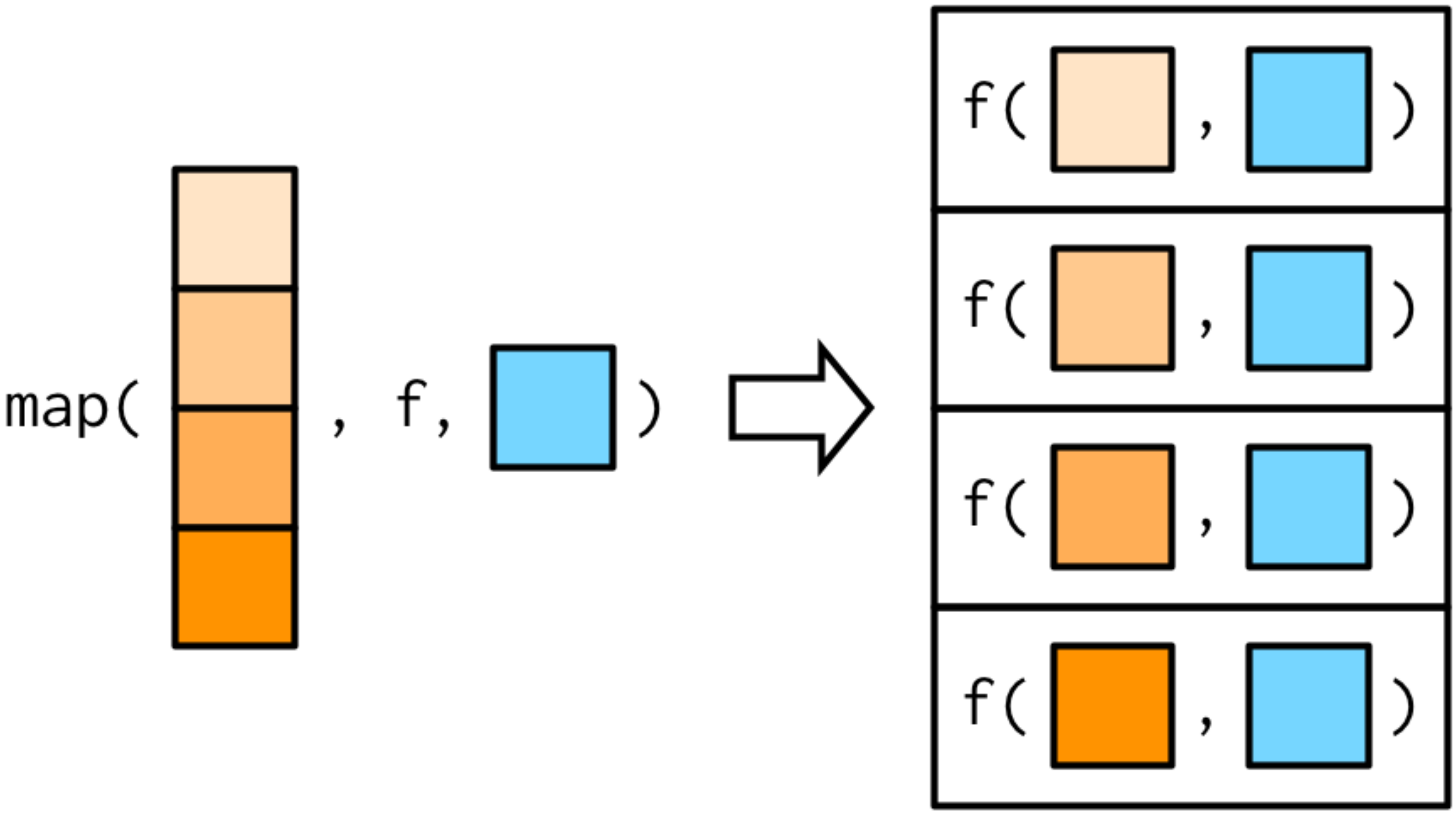
Are either type-unstable (`sapply()`) or verbose (`vapply()`)

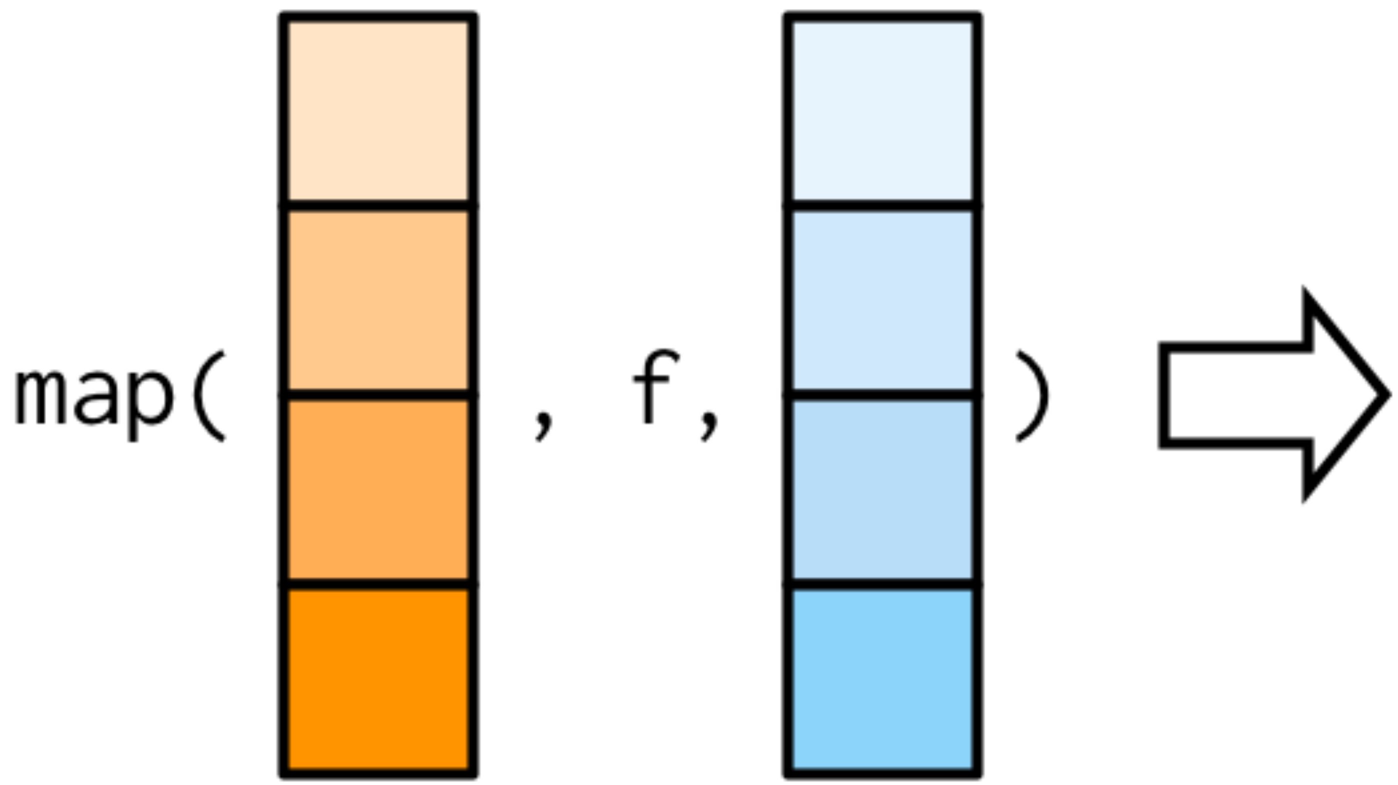
Paired map

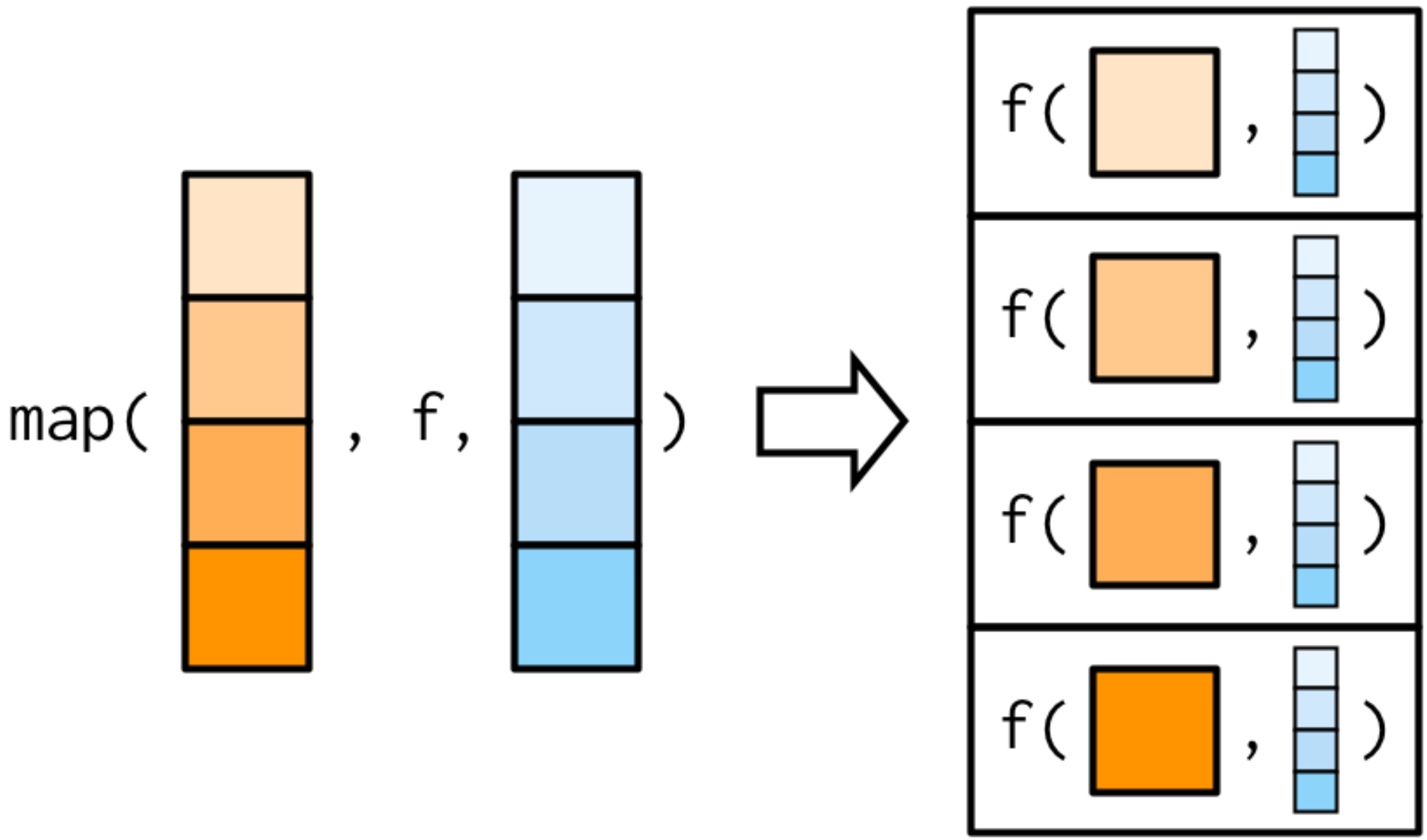


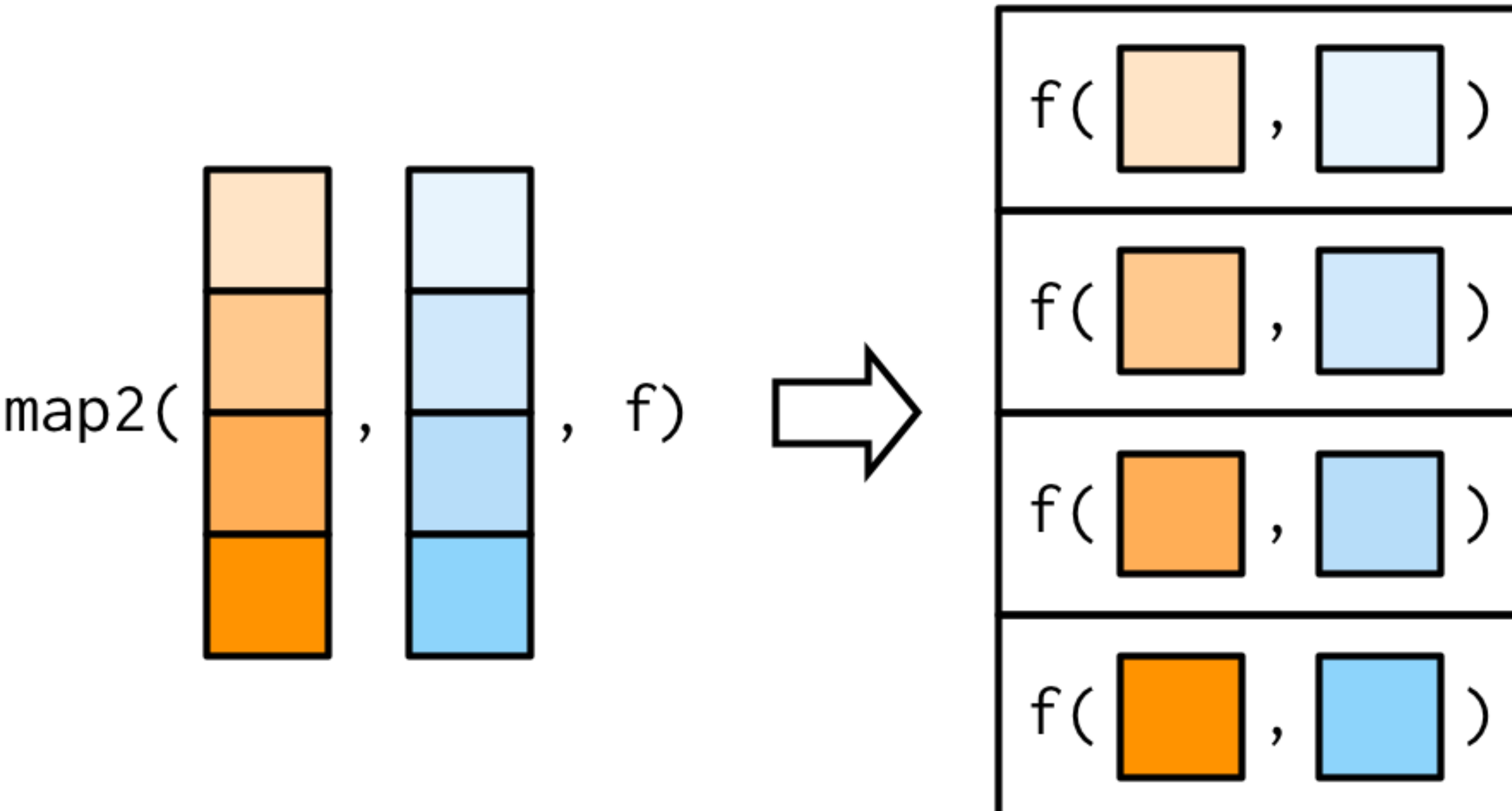
Equivalent to lapply()


map(
, f,
)


map(
, f,
)
f(
,
)
f(
,
)
f(
,
)
f(
,
)


map(
, f,
)


map(
, f,
)
f(
,
)
f(
,
)
f(
,
)
f(
,
)


map2(, , f)
f(,)
f(,)
f(,)
f(,)

When you need to iterate over two objects: map2()

1. Solve for single .x and .y
2. Generalise solution with appropriate map2() function
3. Simplify (if possible)

Goal: save data to paths

```r
library(ggplot2)

# a list of data frames
by_color <- split(diamonds, diamonds$color)

# a vector of paths
paths <- paste0(names(by_color), ".csv")
```

1. Solve for single .x and .y

```
# Solve for one
.x <- by_color[[1]]
.y <- paths[[1]]

write.csv(.x, .y)
```

2. Generalise solution with map2()

```
#                         write.csv(.x, .y)
map2(by_color, paths, ~ write.csv(.x, .y))

# Use more appropriate function
walk2(by_color, paths, ~ write.csv(.x, .y))

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

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

Principle:

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

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
usethis::use_package("purrr", "depends")
```

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

Your turn

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

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

With `R/col_write.R` open you can run `devtools::test_file()` to run only the tests relevant to this file.

A solution

```
col_write <- function(df, path = tempdir()) {
  filenames <- paste0(path, "/", names(df), ".txt")

  walk2(
   df, filenames,
   ~ writeLines(as.character(.x), .y)
  )
}
```

Other types of iteration

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

Type stability

Principle:

Minimise context needed to predict output type

Type depends on:

1. Nothing (constant)
2. Type of first argument
3. Type of another argument

Prefer

4. Types in ...

Avoid

5. Types and order of ...
6. Value of an argument
7. Value of multiple arguments

Place the following functions:

```
mean()
lapply()
sum()
map_int()
ifelse()
c()
median()
sapply()
```

1. Nothing (constant)
2. Type of first argument
3. Type of another argument
4. Types in ...
5. Types and order of ...
6. Value of an argument
7. Value of multiple arguments

Some examples

```
c(Sys.Date(), Sys.time())
c(Sys.time(), Sys.Date())

x <- list(Sys.time(), factor("x"))
sapply(x, class)
sapply(x[1], class)
sapply(x[2], class)

ifelse(TRUE, "x", 1)
ifelse(FALSE, "x", 1)
ifelse(NA, "x", 1)
```

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()
```

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

```r
# In R/col_means.R
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)
```

Principle:

Think about invariants

What should always be true?

What are the invariants?

```
# What should always be true about the output?
# * should be a data frame
expect_s3_class(out, "data.frame")

# * one row
expect_equal(nrow(out), 1)

# * one col for each numeric column in the input
expect_equal(ncol(out), sum(map_lgl(in, is.numeric))
```

sapply and [are not type stable

list or logical vector

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

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

vector or
data frame

One possible solution

```
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 a 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 a 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)
}
```

Failed invariant

```
col_means(data.frame())
#> data frame with 0 columns and 0 rows

# Should be
#> data frame with 0 columns and 1 rows

# Is fixing this important? 🤷‍♂️
```