

Functional programming

Jan 2019

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 == -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 $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ 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 $\frac{2}{3}$ 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.

Chocolate cupcakes

**The
hummingbird
bakery
cookbook**

$\frac{3}{4}$ cup + 2T flour

2 $\frac{1}{2}$ T cocoa powder

a scant $\frac{3}{4}$ cup sugar

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

1 egg

$\frac{1}{4}$ 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 $\frac{2}{3}$ 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**

1 cup flour

Preheat oven to 350°F.

a scant $\frac{3}{4}$ cup sugar

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.

1 $\frac{1}{2}$ t baking powder

3 T unsalted butter

$\frac{1}{2}$ cup whole milk

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.

1 egg

$\frac{1}{4}$ t pure vanilla extract

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 $\frac{2}{3}$ 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

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

140g sugar

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

1.5 t baking powder

Beat in remaining half. Mix until smooth.

40g unsalted butter

Bake 20-25 min at 170°C.

120ml milk

1 egg

0.25 t pure vanilla extract

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.

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.

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

120g flour

140g sugar

1.5t baking powder

40g butter

120ml milk

1 egg

0.25 t vanilla

Chocolate

100g flour

20g cocoa

140g sugar

1.5t baking powder

40g butter

120ml milk

1 egg

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

```
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

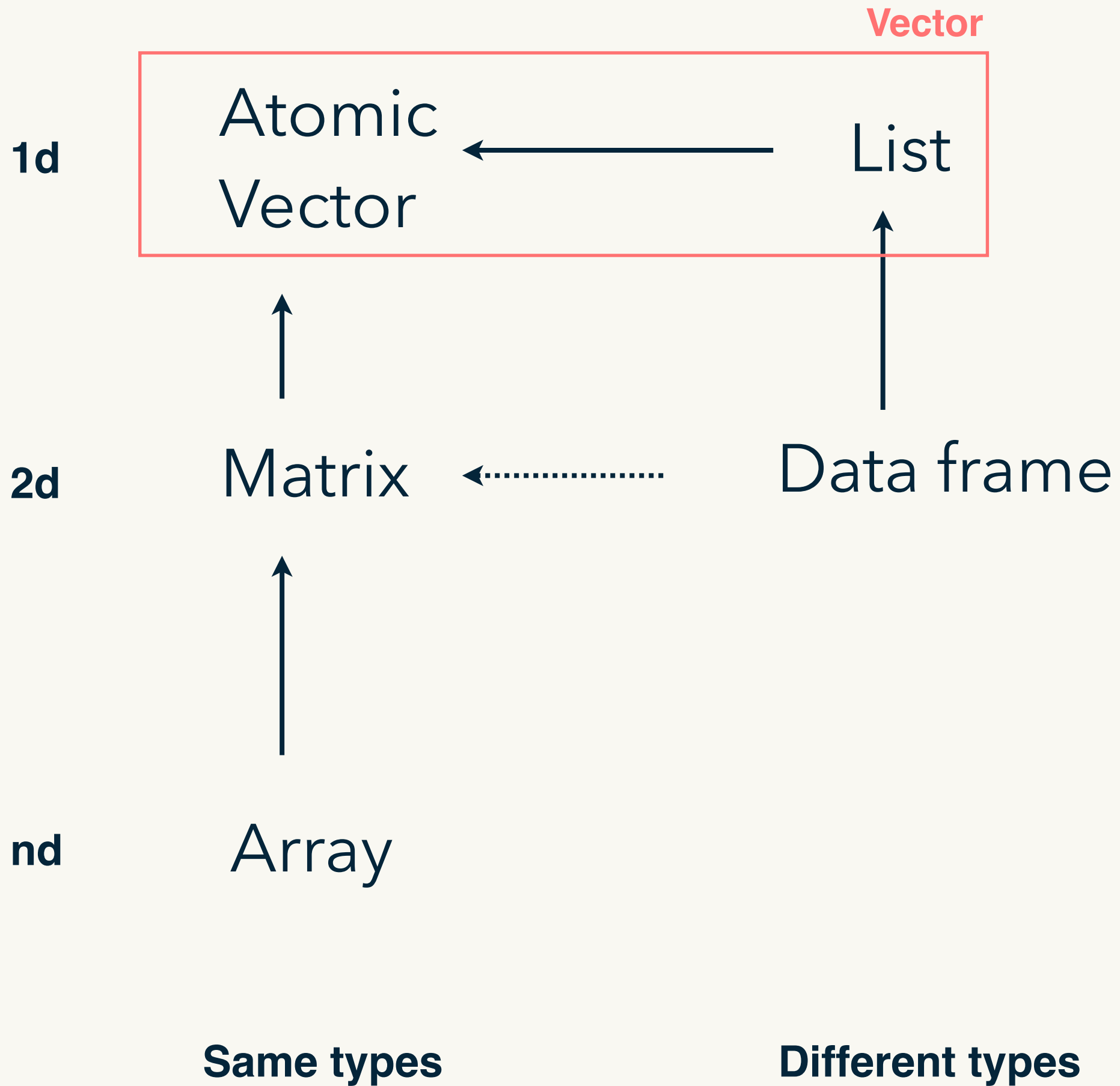
Warmups

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?



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]`

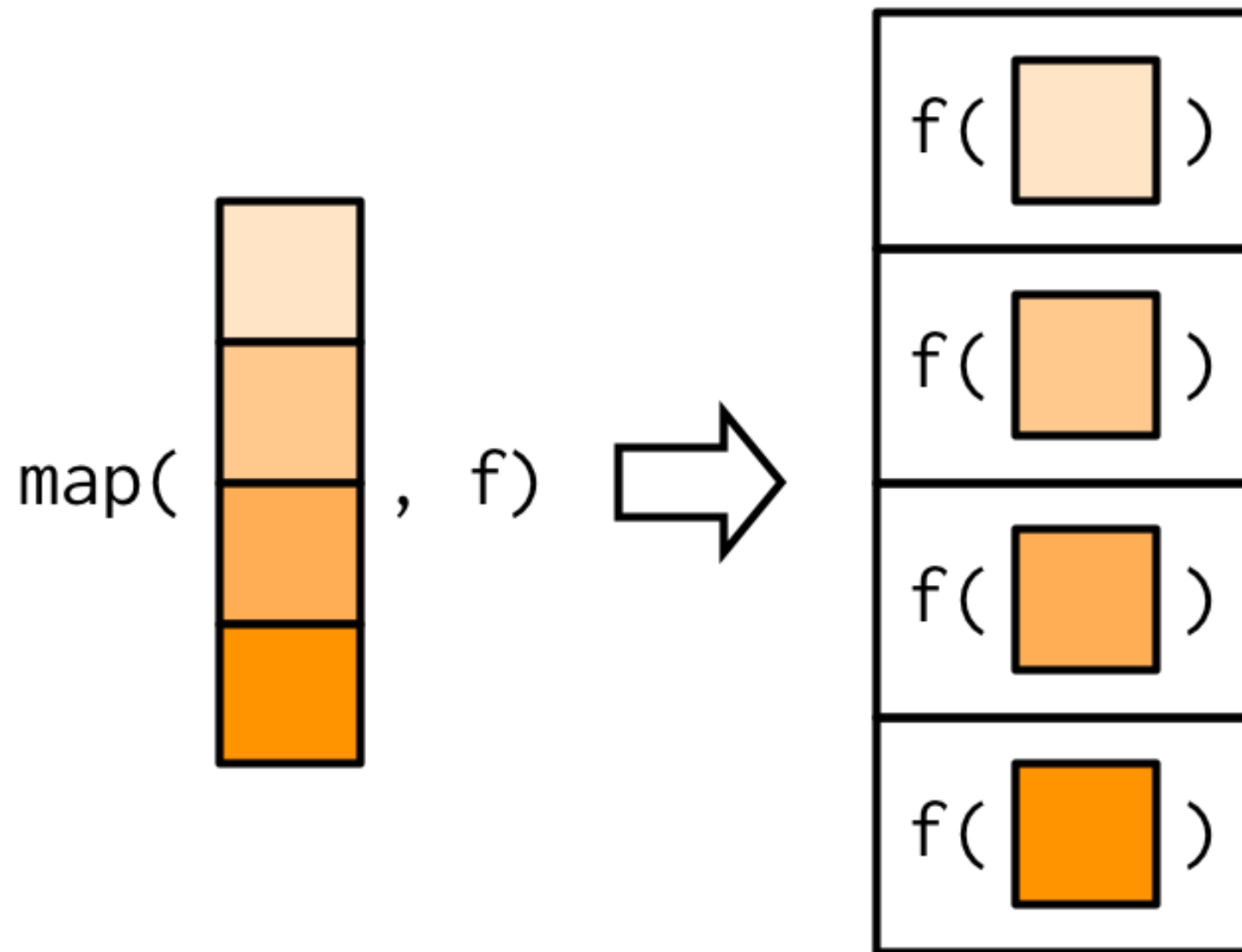


What does this code do?

```
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

map(): for each element, apply f



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

```
strings <- c("a|b", "a|b|c", "d|e", "b|c|d")
```

```
# We want:
```

```
# "a" "a" "d" "b"
```

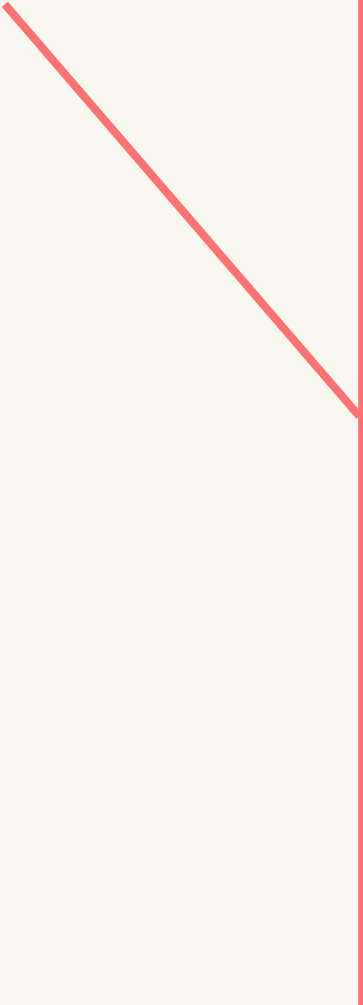
```
# A useful intermediate object
```

```
strings_split <- strsplit(strings, "|", fixed = TRUE)
```

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

For each element of
strings_split,

take it, and extract the first
element

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)

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 double
```

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(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?

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)

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()`)

Base R only provides a partial set of functions

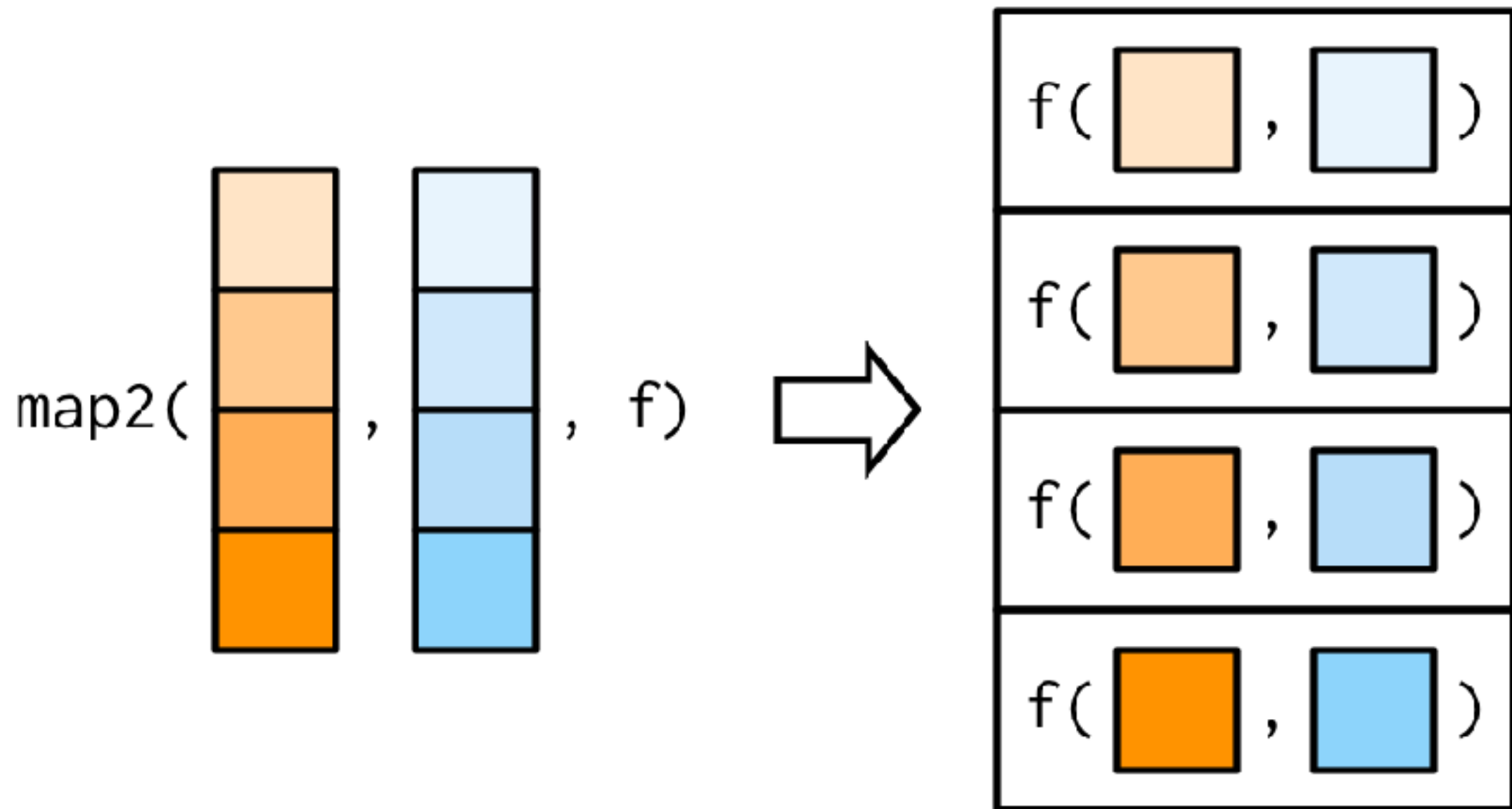
Number of inputs		Output is a scalar	Output is anything	Output is nothing
	1	sapply() / vapply()	lapply()	
	2			
	n	mapply()	Map()	

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()

Paired map

`map2()`: for each pair of elements, apply `f`



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

For an iteration task:

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

Goal: save data to paths

```
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
```

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

Pros

Easily call purrr
functions

Cons

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

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.

`map()`

`sapply()`

`data.frame()`



Returns list, or
dies trying

Output type
depends on input
type, length &
function

Factor vs character
depends on global
setting

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

```
# 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

```
col_means <- function(df) {  
  numeric <- sapply(df, is.logical)  
  numeric_cols <- df[, numeric]  
  
  as.data.frame(lapply(numeric_cols, mean))  
}
```

list or logical vector

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

```
col_means <- function(df) {  
  numeric <- map_lgl(df, is.numeric)  
  numeric_cols <- df[, numeric, drop = FALSE]  
  
  as.data.frame(map(numeric_cols, mean))  
}
```

always a logical
vector

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

Output type
depends on
input type
keep()

map()

sapply()

data.frame()

Returns list, or
dies trying

Output type
depends on input
type, length &
function

Factor vs character
depends on global
setting

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? 🤷
```

Learning More

R4DS: <https://r4ds.had.co.nz/iteration.html>

Advanced R: <https://adv-r.hadley.nz/functionals.html>

Apply functions with purrr : : CHEAT SHEET



Apply Functions

Map functions apply a function iteratively to each element of a list or vector.

`map(x, fun, ...)` → `fun(x)` → `map(x, fun, ...)` Apply a function to each element of a list or vector. *map(x, is.logical)*

`map2(x, y, fun, ...)` → `fun(x, y)` → `map2(x, y, fun, ...)` Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*

`pmap(list(x, y, z), fun, ...)` → `fun(x, y, z)` → `pmap(list(x, y, z), fun, ...)` Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

`invoke_map(x, fun, ...)` → `fun(x)` → `invoke_map(x, fun, ...)` Apply function to each list-element of a list or vector. Run each function in a list. Also `invoke`. *l <- list(var, sd); invoke_map(l, x = 1:9)*

`lmap(x, .f, ...)` Apply function to each list-element of a list or vector. `imap(x, .f, ...)` Apply .f to each element of a list or vector and its index.

OUTPUT

`map()`, `map2()`, `pmap()`, `imap` and `invoke_map` each return a list. Use a suffixed version to return the results as a specific type of flat vector, e.g. `map2_chr`, `pmap_lgl`, etc.

Use `walk`, `walk2`, and `pwalk` to trigger side effects. Each return its input invisibly.

function	returns
<code>map</code>	list
<code>map_chr</code>	character vector
<code>map_dbl</code>	double (numeric) vector
<code>map_dfc</code>	data frame (column bind)
<code>map_dfr</code>	data frame (row bind)
<code>map_int</code>	integer vector
<code>map_lgl</code>	logical vector
<code>walk</code>	triggers side effects, returns the input invisibly

SHORTCUTS - within a purrr function:

"name" becomes `function(x) x[["name"]]`, e.g. `map(l, "a")` extracts `a` from each element of `l`

`~.x` becomes `function(x) x`, e.g. `map(l, ~.x)` becomes `map(l, function(x) x)`

`~.x.y` becomes `function(x, y) .x.y`, e.g. `map2(l, p, ~.x+y)` becomes `map2(l, p, function(l, p) l + p)`

`~.1..2` etc becomes `function(..1, ..2, etc) ..1..2` etc, e.g. `pmap(list(a, b, c), ~.3 + ..1 - ..2)` becomes `pmap(list(a, b, c), function(a, b, c) c + a - b)`

Work with Lists

FILTER LISTS

`pluck(x, ..., .default=NULL)` Select an element by name or index, `pluck(x, "b")`, or its attribute with `attr_getter`. `pluck(x, "b", attr_getter("n"))`

`keep(x, .p, ...)` Select elements that pass a logical test. `keep(x, is.na)`

`discard(x, .p, ...)` Select elements that do not pass a logical test. `discard(x, is.na)`

`compact(x, .p = identity)` Drop empty elements. `compact(x)`

`head_while(x, .p, ...)` Return head elements until one does not pass. Also `tail_while`. `head_while(x, is.character)`

RESHAPE LISTS

`flatten(x)` Remove a level of indexes from a list. Also `flatten_chr`, `flatten_dbl`, `flatten_dfc`, `flatten_dfr`, `flatten_int`, `flatten_lgl`, `flatten(x)`

`transpose(l, .names = NULL)` Transposes the index order in a multi-level list. `transpose(x)`

SUMMARISE LISTS

`every(x, .p, ...)` Do all elements pass a test? `every(x, is.character)`

`some(x, .p, ...)` Do some elements pass a test? `some(x, is.character)`

`has_element(x, .y)` Does a list contain an element? `has_element(x, "foo")`

`detect(x, .f, ..., .right=FALSE, .p)` Find first element to pass. `detect(x, is.character)`

`detect_index(x, .f, ..., .right=FALSE, .p)` Find index of first element to pass. `detect_index(x, is.character)`

`vec_depth(x)` Return depth (number of levels of indexes). `vec_depth(x)`

JOIN (TO) LISTS

`append(x, values, after = length(x))` Add to end of list. `append(x, list(d = 1))`

`prepend(x, values, before = 1)` Add to start of list. `prepend(x, list(d = 1))`

`splice(...)` Combine objects into a list, storing S3 objects as sub-lists. `splice(x, y, "foo")`

TRANSFORM LISTS

`modify(x, .f, ...)` Apply function to each element. Also `map`, `map_chr`, `map_dbl`, `map_dfc`, `map_dfr`, `map_int`, `map_lgl`. `modify(x, ~.+2)`

`modify_at(x, .at, .f, ...)` Apply function to elements by name or index. Also `map_at`. `modify_at(x, "b", ~.+2)`

`modify_if(x, .p, .f, ...)` Apply function to elements that pass a test. Also `map_if`. `modify_if(x, is.numeric, ~.+2)`

`modify_depth(x, depth, .f, ...)` Apply function to each element at a given level of a list. `modify_depth(x, 1, ~.+2)`

WORK WITH LISTS

`array_tree(array, margin = NULL)` Turn array into list. Also `array_branch`. `array_tree(x, margin = 3)`

`cross2(x, y, .filter = NULL)` All combinations of `x` and `y`. Also `cross`, `cross3`, `cross_dfc`. `cross2(1:3, 4:6)`

`set_names(x, nm = x)` Set the names of a vector/list directly or with a function. `set_names(x, c("p", "q", "r"))` `set_names(x, tolower)`

Reduce Lists

`reduce(x, .f, ..., .init)` Apply function recursively to each element of a list or vector. Also `reduce_right`, `reduce2`, `reduce2_right`. `reduce(x, sum)`

`accumulate(x, .f, ..., .init)` Reduce, but also return intermediate results. Also `accumulate_right`. `accumulate(x, sum)`

Modify function behavior

`compose()` Compose multiple functions.

`lift()` Change the type of input a function takes. Also `lift_dbl`, `lift_dv`, `lift_ld`, `lift_lv`, `lift_vd`, `lift_vl`.

`rerun()` Rerun expression n times.

`negate()` Negate a predicate function (a pipe friendly !)

`partial()` Create a version of a function that has some args preset to values.

`safely()` Modify func to return list of results and errors.

`quietly()` Modify function to return list of results, output, messages, warnings.

`possibly()` Modify function to return default value whenever an error occurs (instead of error).



<https://github.com/rstudio/cheatsheets/raw/master/purrr.pdf>

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/)