Lecture 24: Nov 7, 2018

Functionals

- *Ubiquitousness of Functions*
- *Environments*
- *Overview of Functionals*
- *Functionals in Practice*
- *An Odyssey in purrr*

James Balamuta
STAT 385 @ UIUC

Announcements

- **Group Proposal** due **Friday, November 16th** at **11:59 PM**
- **EC Opportunity**: Big Data Summit, Nov 8th
  - https://github.com/stat385-fa2018/disc/issues/71
- **Quiz 11** covers Week 10 contents @ CBTF.
  - Window: Nov 6th - Nov 8th
  - Sign up: https://cbtf.engr.illinois.edu/sched
- Want to review your homework or quiz grades? **Schedule an appointment.**

Lecture Objectives

- Understanding that functions are universal
- Describe the three tenets of functional programming
- Explain how Split-Apply-Combine is viewed in a functional paradigm.
- Applying functional programming within *R*

Ubiquitousness of Functions

Functions are the **lingua franca**

... common language of all programming languages ...

R Foundations

"Everything that *exists* in *R* is an **Object**.
Everything that *happens* in *R* is a **Function Call**.
Interfaces to other software are part of *R*"

–John M. Chambers, Extending R (2016) pg. 4

From SQL &
more during HPC

On Today's Agenda

Functions are Objects

... pulling out function definitions ...

```r
# Create a square function
square = function(x) {
  x^2
}

# Find high-level class
# information
class(square)
# [1] "function"

# Obtain low-level class
# information
typeof(square)
# [1] "closure"
```

Extracting Properties

... pulling out function definitions ...

```r
# Retrieve parameters &
# default values
formals(square)
# $x

# Retrieve the function body
body(square)
# {
#    x^2
# }

# Retrieve the location of
# function
environment(square)
# <environment:
R_GlobalEnv>
```

Hidden Function Calls

Addition

```r
# Everything that happens in
# R is a Function Call.

10 + 25
# [1] 35

`+`(10, 25)
# [1] 35
```

Assignment

```r
x = c(1, 2, 3)

`=`(x, c(1, 2, 3))
```

Subset

```r
x[1]
# [1] 1

`[`(x, 1)
# [1] 1
```

Anonymous Functions

… a failure to name and lambda functions …

```
function(x = 4) { x + 1 }                  # No Name Function
# function(x = 4) x + 1
# <environment: 0x7fad925f1298>

(function(x = 4) { x + 1 })(2)            # Anonymous definition
# [1] 3

add_one = function(x = 4) { x + 1 } # Named function
add_one(2)
# [1] 3
```

Function as a Parameter

... changing operations ...

```r
# Define operations
add = function(x, y) { x + y }
subtract = function(x, y) { x - y }
multiply = function(x, y) { x * y }

# Create a function that
# calls others
do_operation = function(f, x, y) {
  f(x, y)
}

# Perform add operation
do_operation(add, 2, 5)
# [1] 7

# Perform subtract operation
do_operation(subtract, 2, 5)
# [1] -3
```

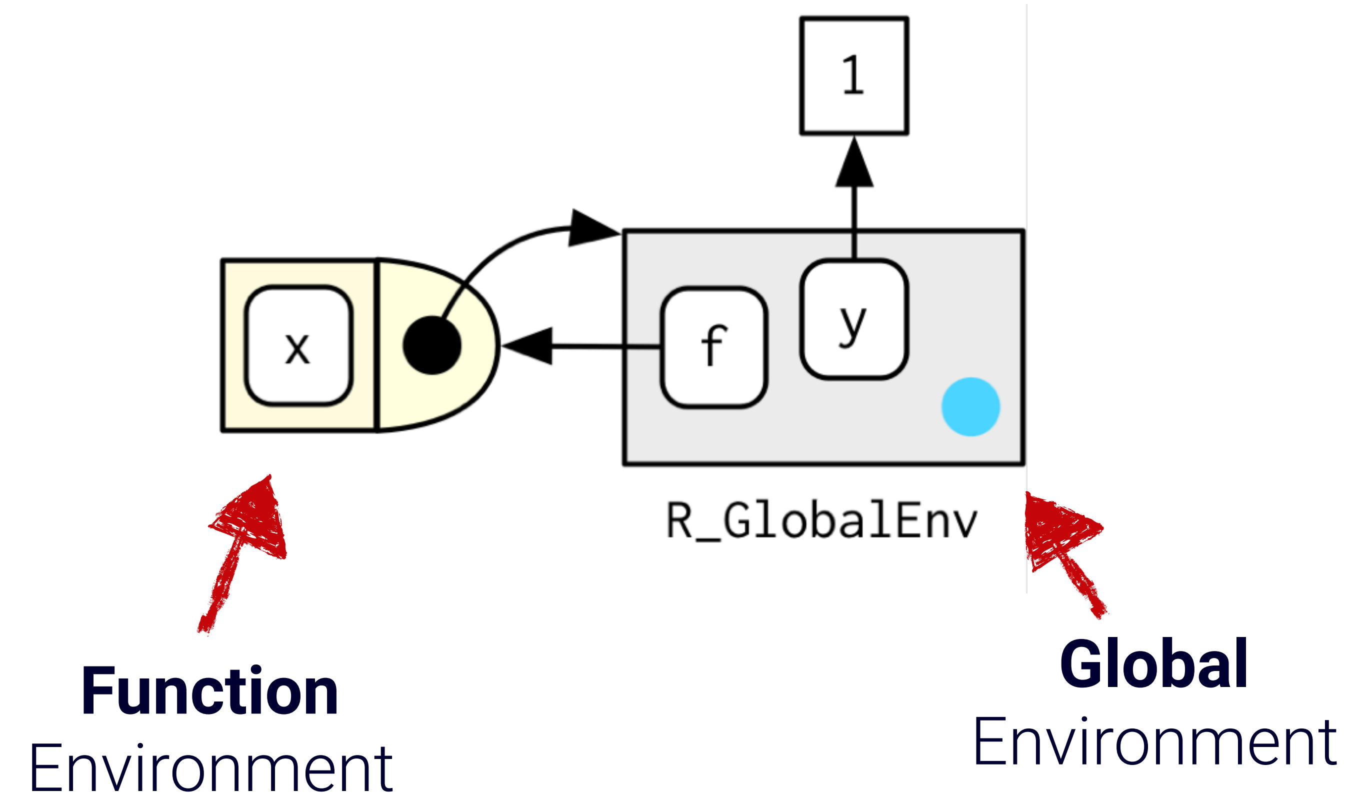
Your Turn

Determine the function properties of **mean()**

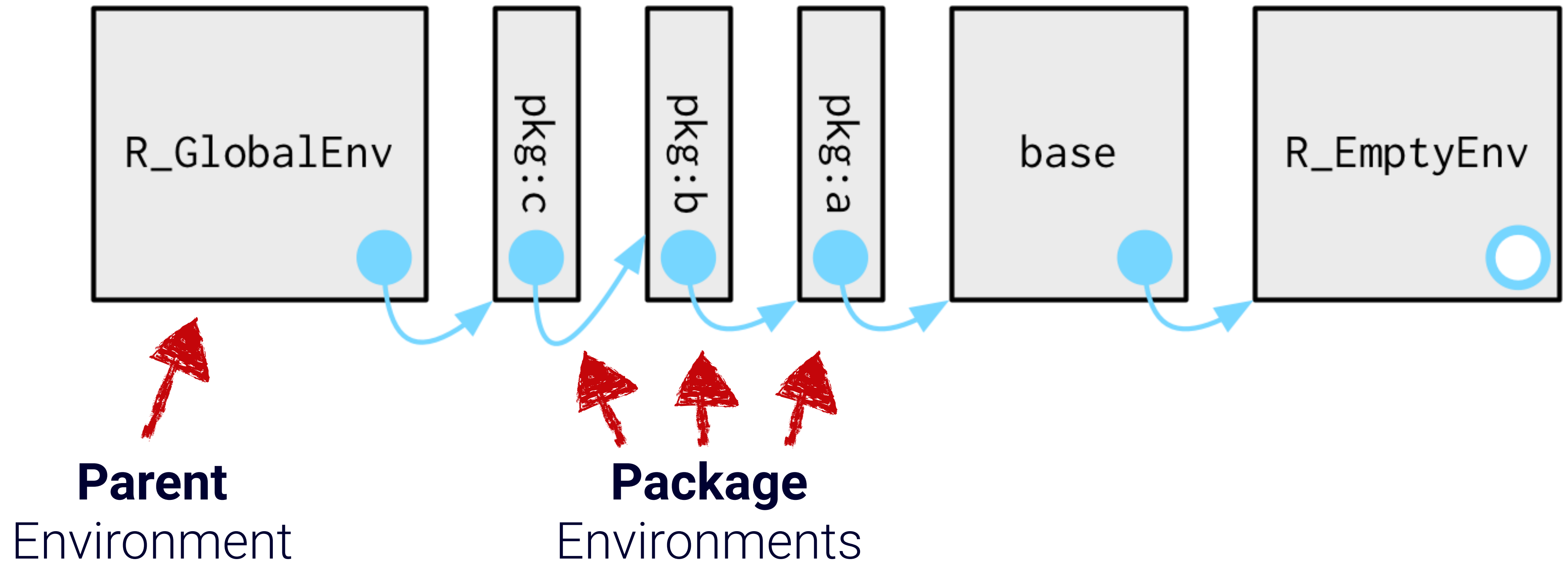
Environments

Definition:
Environments refer to where variable or function information reside.

```
# Global Environment
y = 1
f = function(x) {
  # Function Environment
  x + y
}
```



Definition:
Scope refers to the rules for accessing and using values or functions within an environment.



```r
# Note `value` has not been
# defined.
multiple_constant =
function(x) {
  return(value * x)
}

# Only on call is an error
# detected.
multiple_constant(5)

## Error in
multiple_constant(5) :
## object 'value' not found
```

Environment Scope

… use of variables within a function …

```r
# Define value in global
# environment (e.g. outside of
# the function)
value = 3

# Create a function that uses
# the global value
multiple_constant =
 function(x) {
    # Note: `value` is not defined
    # in the function body or as
    # a parameter.
    return(value * x)
}

# Call the function
multiple_constant(5)

# What comes out?
# What does this say about where
# variables reside?
```

Local v Global Environments

... *R*'s scoping of values ...

Your Turn

Spot the error in the function given below

```
# Create some data
x = rnorm(10)
n = length(x)

# Define a function for mean
my_mean = function(x) {
   summed = 1/n * sum(x)
   summed
}

my_mean(x)
```

Overview of Functionals

Functional Programming

... three tenets ...

1. Functions are **first-class** objects ...
   ... can be stored as *variables* ...

2. Functions are **higher-order** ...
   ... accept a function as argument, return a function, or both....

3. **Closures** ...
   ... functions returned with an external scope ...

Definition:
Functionals or *higher-order functions* are functions whose *input* takes a function, operates on it, and then returns the resulting output.

functional(input-vector , function) → output-vector

$$\text{functional}\left(\begin{bmatrix} \; \\ \; \\ \; \\ \; \end{bmatrix}, f\right) \to f\left(\begin{bmatrix} \; \\ \; \\ \; \\ \; \end{bmatrix}\right)$$

Functionals in Practice

... recipe applied multiple times ...

```r
# Create a function that
# can call another function
call_func = function(x, f) {
  # call the function `f`
  # with data `x`
  f(x)
}

# Sample data
x = c(-2, 0.3, 1.2, 4.8)

# Compute the mean
call_func(x, mean)
# [1] 1.075

# Compute the min
call_func(x, min)
# [1] -2
```

Simulations

… splitting by iteration …

```r
# Set a seed for reproducibility
set.seed(111)

# Call function 3 times
replicate(3, runif(5))
#           [,1]       [,2]       [,3]
# [1,] 0.5929813 0.41833733 0.55577991
# [2,] 0.7264811 0.01065785 0.59022849
# [3,] 0.3704220 0.53229524 0.06714114
# [4,] 0.5149238 0.43216062 0.04754785
# [5,] 0.3776632 0.09368152 0.15620252

# Repeat same result 3 times
rep(runif(5), 3)
# [1]  0.4464278 0.1714437 0.9665343
# [4]  0.3106664 0.6144664 0.4464278
# [7]  0.1714437 0.9665343 0.3106664
# [10] 0.6144664 0.4464278 0.1714437
# [13] 0.9665343 0.3106664 0.6144664
```

Definition:
Ellipsis or *dot-dot-dot* (...) allow for any number of parameters to be passed in to the function being called.

```
call_func = function(x, f, ... ) {
  f(x, ... )
}

x[c(1, 3)] = NA                    # Impute NA values into the vector
x
# [1]  NA 0.3  NA 4.8

call_func(x, min, na.rm = FALSE) # Default behavior of min()
# [1] NA
call_func(x, min, na.rm = TRUE)  # Pass a new parameter
# [1] 0.3
```

Ellipsis in Practice

```r
?paste

# Infinite number of string
# combinations

paste("first", 1, "second", 8)


?data.frame

# Infinite number of columns
# of any type allowed

data.frame(
    x = 1, y = 1:10
)
```

Your Turn

Use the **replicate()** function to sample *10 observations* from a normal distribution *5 times*.

Functionals in Practice

Motivating Example hw03

stat385-fa2018 / disc Private

Unwatch 4

Code | Issues 10 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Change multiple columns of pima #38

Closed darrenmuliawan opened this issue on Sep 19 · 4 comments

commented on Sep 19 • edited

I have a question regarding changing multiple columns of data frame at once
`data[, c("column1")] = function(data[, c("column1")], x)`
Why R lets me do this but not this?
`data[, c("column1", "column2", "column3", ...)] = function(data[, c("column1", "column2", "column3", ...)], x)`

This is the error message:
`## Warning in` [<-.data.frame`(`*tmp*`, , c("glucose", "diastolic",

"triceps", : provided 3840 variables to replace 5 variables`

Thank you

commented on Sep 19

Try examining your function by testing it with the same data frame (without assigning it) and see if you can figure out why :)

coatless commented on Sep 20

The issue here is the function isn't setup to handle a `data.frame` containing values. When operating with `ifelse()` we make the explicit assumption that we are using vectors.

To modify the function so that this is possible, we'll need to use a functional that allows us to treat each column of the `data.frame` separately inside of the function call. From the `*apply` family of functions, we'll use `lapply`.

```
data[, c("column1", "column2", "column3", ...)] =
    lapply(data[, c("column1", "column2", "column3", ...)], FUN = function, x =
```

Specifying Missingness

... set a value to be missing ...

```
# Code a value as being missing
my_df$col1[my_df$col1 == -1] = NA
my_df$col2[my_df$col2 == -1] = NA
my_df$col3[my_df$col3 == -1] = NA
my_df$col4[my_df$col4 == -1] = NA
```

Functionize it!

… common pattern -> abstract logic …
… creating a recipe …

```r
# Action repeated consistently
code_missing = function(x, value = -1) {
  x[x == value] = NA
  x
}

# Apply action to data
my_df$col1 = code_missing(my_df$col1)
my_df$col2 = code_missing(my_df$col2)
my_df$col3 = code_missing(my_df$col3)
my_df$col4 = code_missing(my_df$col4)
```

Repeatedly Applying

… recipe applied multiple times …

```r
# Action repeated consistently
code_missing = function(x) {
  x[x == -1] = NA
  x
}

# Apply the behavior uniformly
# to columns
for(i in seq_len(ncol(my_df))) {
  my_df[, i] = code_missing(my_df[, i])
}
```

Emphasis of Repeat

… what is being repeated ???

```r
# Apply the behavior uniformly to columns
for(i in seq_len(ncol(my_df)) {
  my_df[, i] = code_missing(my_df[, i])
}
```

Emphasis of Repeat

… why are we focused on the **object** and **position** ???

```
# Apply the behavior uniformly to columns
for(i in seq_len(ncol(my_df)) {
  my_df[, i] = code_missing(my_df[, i])
}
```

Emphasis of Repeat

… why not the **action** ???

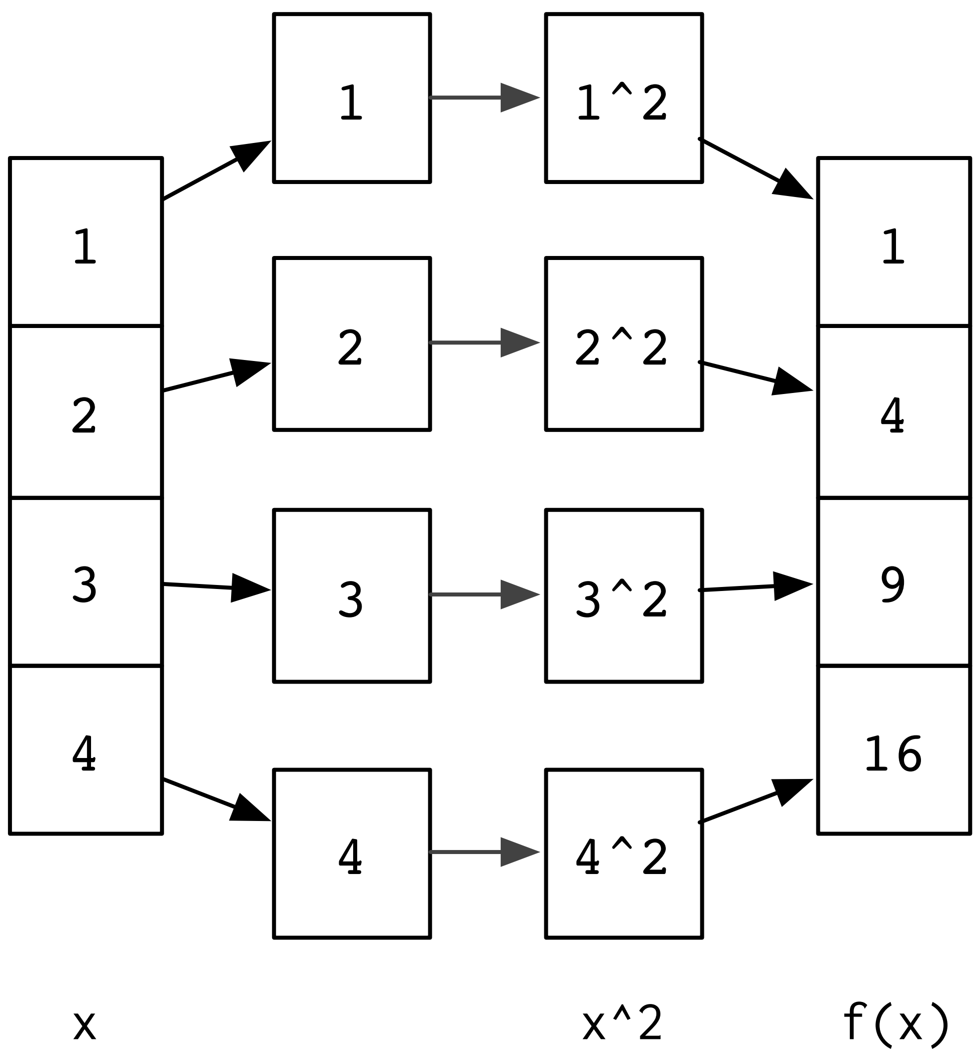
```r
# Apply uniformly the behavior to columns
for(i in seq_len(ncol(my_df)) {
   my_df[, i]= code_missing(my_df[, i])
}
```

Why aren't we emphasizing the **action** over the *object*?

R's View of Objects

... **objects** act as collections of values that have *actions* applied to them ...

```
x = 1L:4L
x^2          # f(x) = x^2
```



How can we replace $f(x) = x^2$ with a **generic function**?

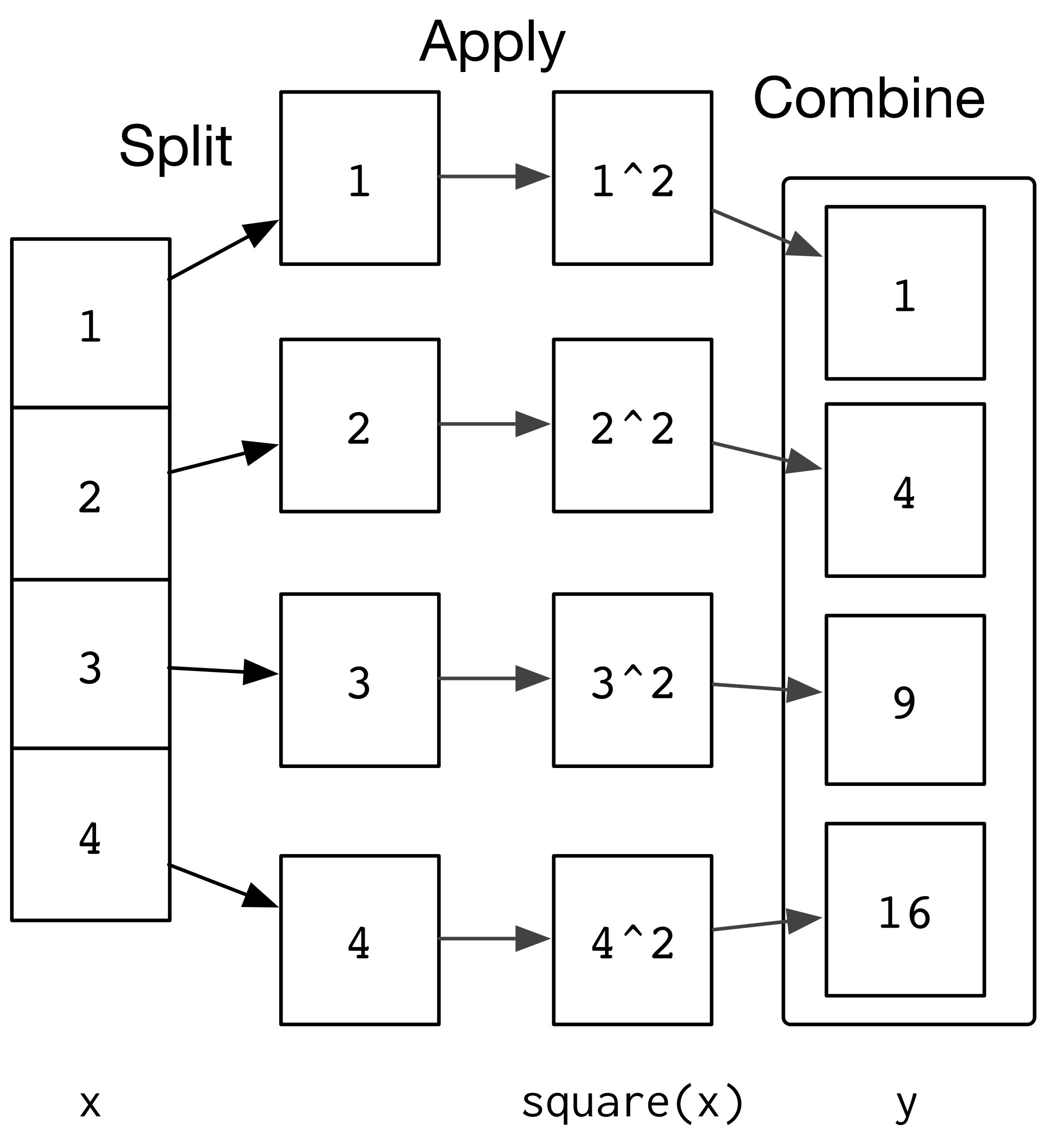
Common Functionals

… functionals in *R* …

Function	Description	Output
sapply	Apply a Function over a List or Vector	vector, matrix, array, list
lapply	Apply a Function over a List or Vector	list
vapply	Apply a Function with type stability over a list or vector	vector
apply	Apply Functions Over Array Margins	vector, matrix
mapply	Apply a Function to Multiple List or Vector Arguments	vector, matrix, array, list
tapply	Apply a Function to Groups	vector

Functionals

... a fun computation example ...




```r
# Sample data
x = c(-2, 0.3, 1.2, 4.8)

# Define function
square = function(x) { x^2 }

# List Output
lapply(x, FUN = square)
# [[1]]
# [1] 1
# [[2]]
# [1] 4
# ...

# Vector / Matrix Output
sapply(x, FUN = square)
# [1]  1  4  9 16
```

Functionals as Loops

… writing our own lapply …

```
# Functional to mimic lapply()
my_lapply = function(x, func) {

 # Setup storage that is a list
 out = vector('list', length(x))

 # Iterate over each element
 for(i in seq_along(out)) {
  # Apply the function to x
  out[[i]] = func(x[[i]])
 }

 out
}

# Check output
my_lapply(x, func = square)
# [[1]]
# [1] 1
# [[2]]
# [1] 4
# ....
```

Iteration vs. Functionals

... functionals emphasize **action** ...

```r
# Obtain the mean of each variable
means = vector("double", ncol(trees))
for(i in seq_along(trees)) {
  means[[i]] = mean(trees[[i]])
}

# Obtain the sd of each variable
sds = vector("double", ncol(trees))
for(i in seq_along(trees)) {
  sds[[i]] = sd(trees[[i]])
}

means
sds
```

```r
# Obtain the mean of each variable
means = sapply(trees, FUN = mean)

# Obtain the sd of each variable
sds = sapply(trees, FUN = sd)

means
sds
```

“Of course someone has to write loops.
It doesn’t have to be you.”

–Jennifer Bryan, UBC, RStudio

```r
# Compute the means
my_means = numeric(length(x))
for (i in seq_along(x)) {
  my_means[[i]] = mean(x[[i]])
}
my_means

# Compute the st. dev
my_sds = numeric(length(x))
for (i in seq_along(x)) {
  my_sds[[i]] = sd(x[[i]])
}
my_sds

# Psst, here's a hint:
# compute_value =
#    function( x , func )  {
#    # content
# }
```

Your Turn

Using functionals, determine the best way to reduce the level of duplication in the code by writing a function.

Apply on Rows

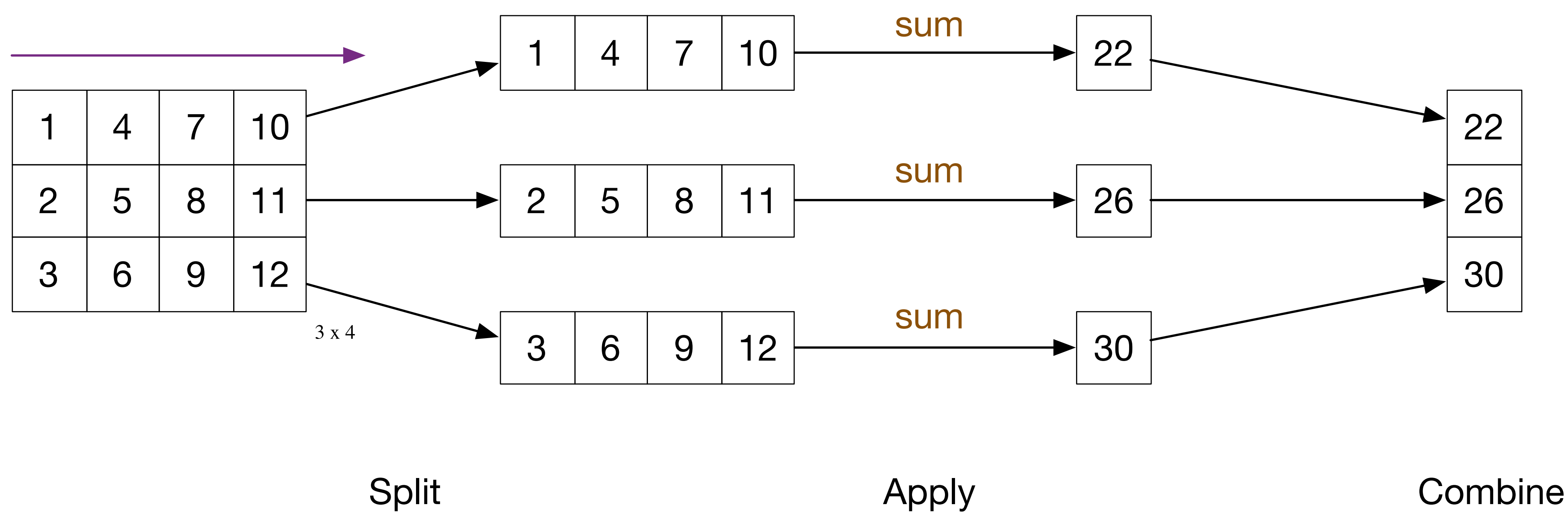
… applying a function to just rows …

Data Structure
Object to be iterated over by the higher-order function

Iteration Control
Dimension to iteratively process over either row (1), column (2), or both c(1, 2)

Function
The operation applied to each item in the defined grouping

```
matrix_row_sum = apply(input_matrix, MARGIN = 1, FUN = sum)
```



Apply on Columns

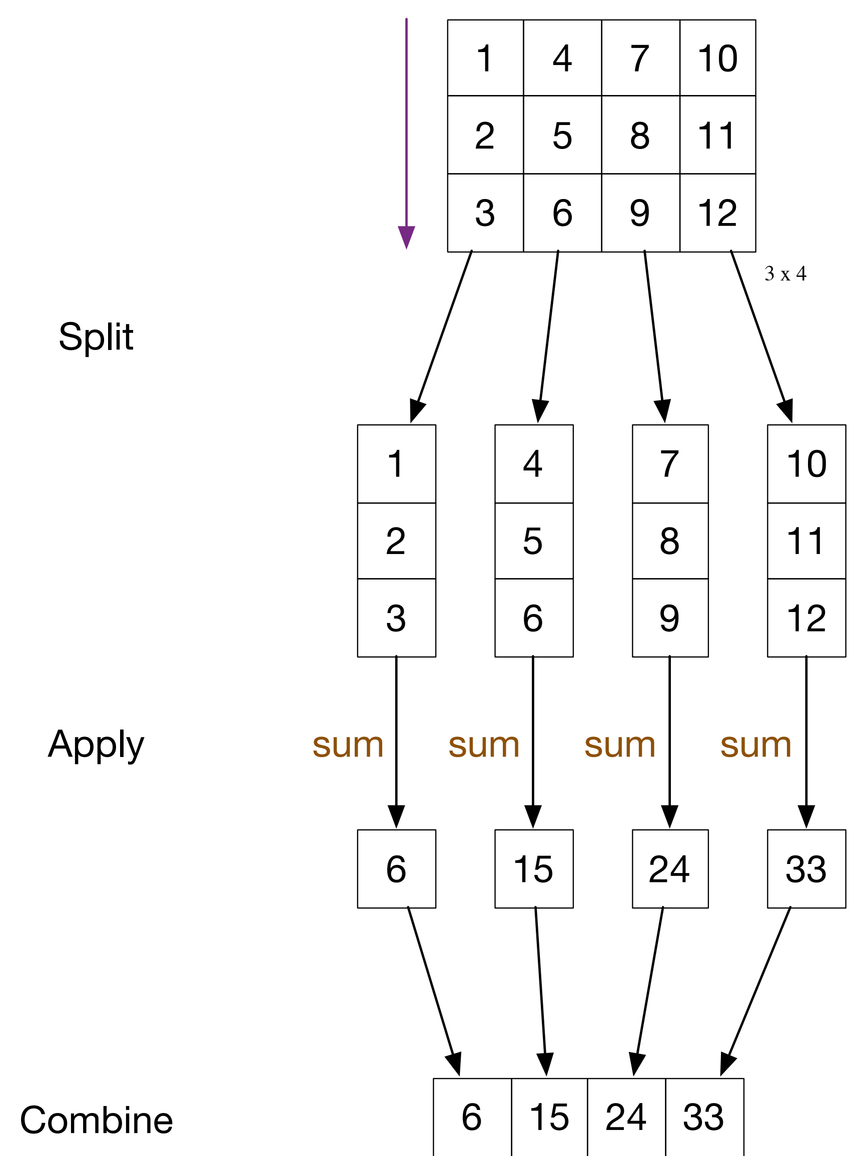
... apply with margin = **2** gives **columns** ...

Data Structure
Object to be iterated over by the higher-order function

Iteration Control
Dimension to iteratively process over either row (1), column (2), or both c(1, 2)

Function
The operation applied to each item in the defined grouping

```
matrix_col_sum = apply(input_matrix, MARGIN = 2, FUN = sum)
```



Apply on Rows + Columns

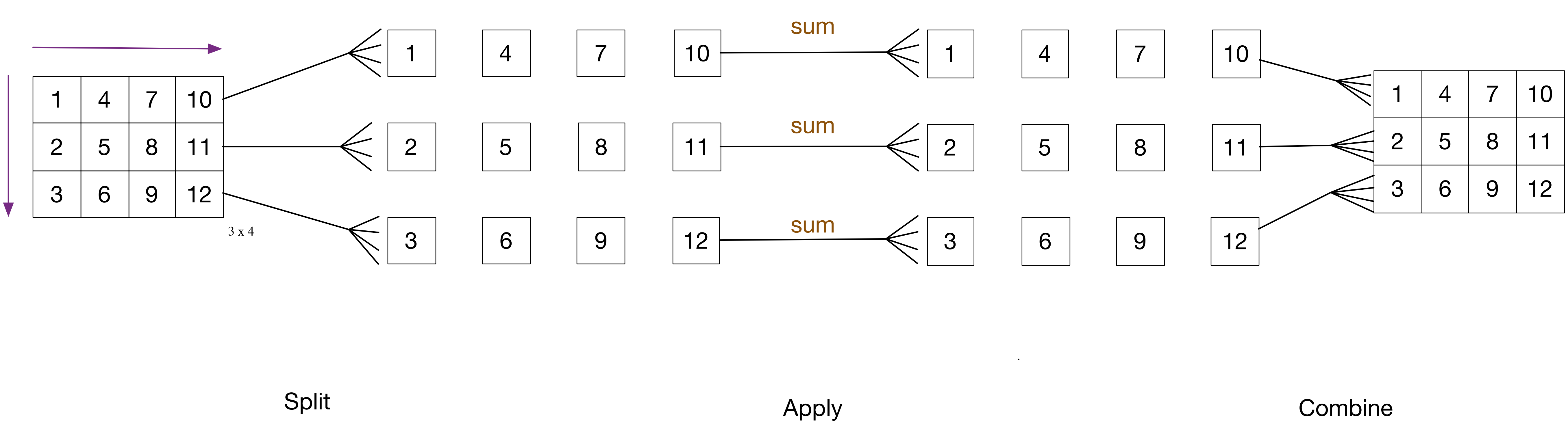
… apply over both rows and columns on a matrix …

Data Structure
Object to be iterated over by the higher-order function

Iteration Control
Dimension to iteratively process over either row (1), column (2), or both c(1, 2)

Function
The operation applied to each item in the defined grouping

```
matrix_element_sum = apply(input_matrix, MARGIN = c(1, 2), FUN = sum)
```



Functions as Data

… power of functionals …

```r
# List containing functions
stat_funs =
 list(min = min, median = median,
        mean = mean, sd = sd,  max = max)

# Apply a Function over a List or Vector
version_one =
   sapply(stat_funs,
      FUN =
        function(x, data) { sapply(data, x) },
      data = trees)

# Apply a Function to Multiple Lists/Vectors
version_two =
     mapply(sapply, stat_funs,
                MoreArgs = list(X = trees))

# Verify approaches are equivalent
all.equal(version_one, version_two)
# [1] TRUE
```

Your Turn

1. Determine the classes of **mtcars** with **class()**

2. Use the **summary()** on three data sets:

```
data_combined = list(PlantGrowth, rock, mtcars)
```

3. Compute the quantiles for the data in two ways:
   using a **for** loop and a functional.

```
sim_data = list(normal_nums = rnorm(100),
                uniform_nums = runif(50))
```

An Odyssey in purrr

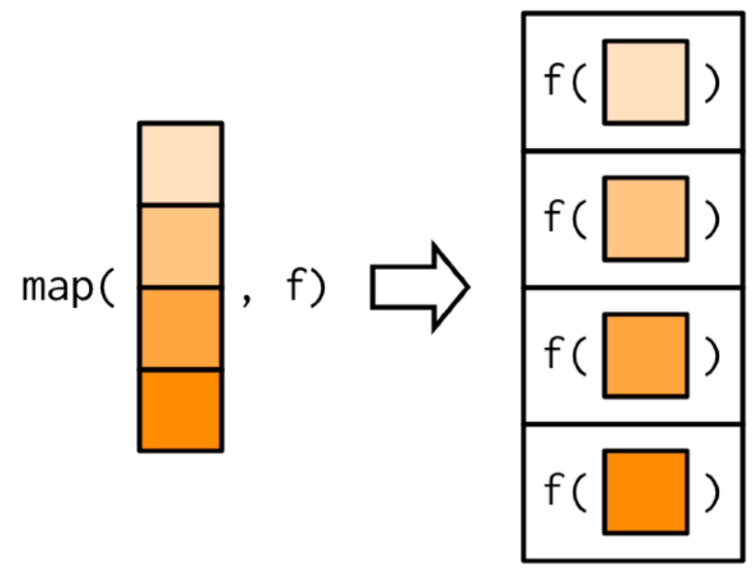
... a functional programming toolkit for R ...

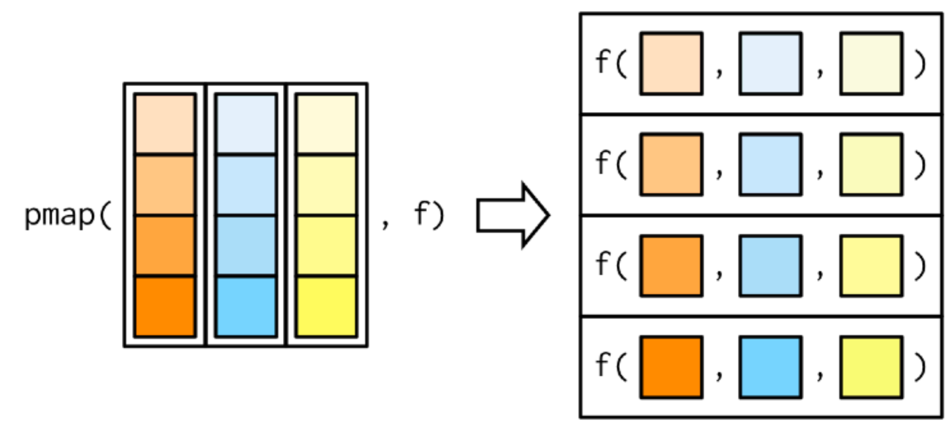
```r
install.packages("purrr")
library("purrr")
```

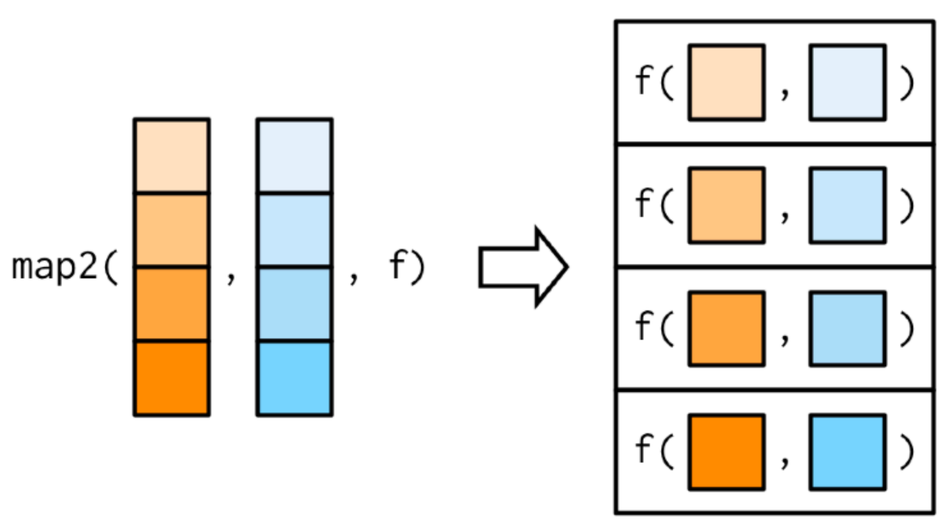
Dim Input	Scalar	List	Side-effects
1	map_lgl() map_int() map_dbl() map_chr()	map()	walk()
2	map2_lgl() map2_int() map2_dbl() map2_chr()	map2()	walk2()
n	pmap_lgl() pmap_int() pmap_dbl() pmap_chr()	pmap()	pwalk()

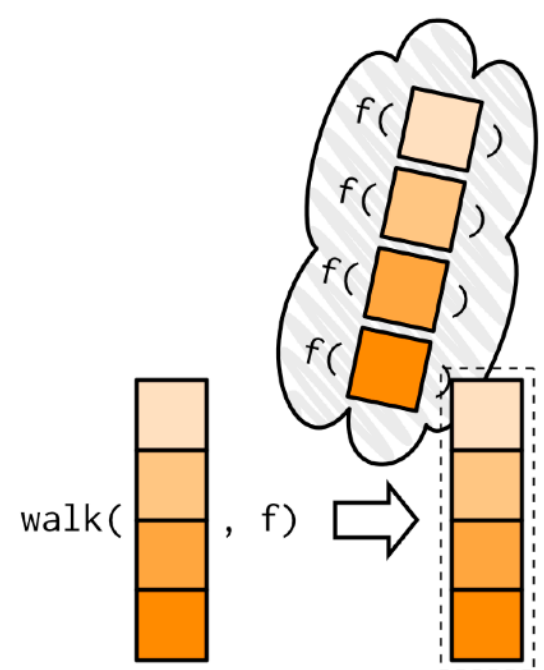
Graphical Overview

... how each purrr functionals work ...







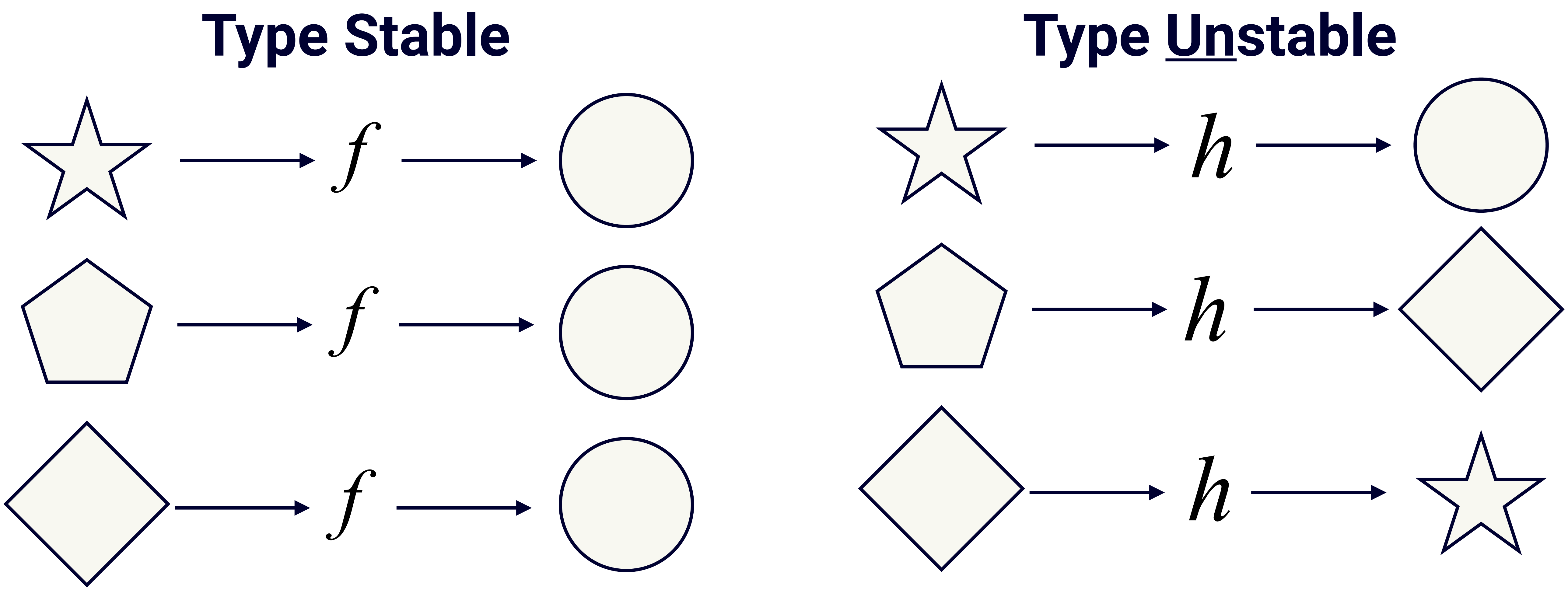


purrr to base

purrr	Base R	Description
map()	lapply()	List output
map_*()	vapply()	Type Stable
map2() / pmap()	mapply()	Multiple Inputs

Definition:

Type Stability refers to the return value of functions being consistent regardless of the input.



Type Unstable Functions

"My mama always said, life was like a box of chocolates. You *never* know what you're gonna get."

– Forrest Gump

purrr type-stability

... example ...

```r
library("purrr")

# Map output to list
map(mtcars, mean)

# Using base R
lapply(mtcars, FUN = mean)

# Map to double vector
map_dbl(mtcars, mean)

# Base R type-stable map
vapply(mtcars,
   FUN = mean,
   FUN.VALUE = numeric(1))

# Avoid using type-unstable map
sapply(mtcars, FUN = mean)
```

Recap

- **Ubiquitousness of Functions**
  - *R* is a functional language
  - Functions are objects
- **Environments**
  - Indicate where information is stored.
- **Functionals**
  - Functionals use a function input with a vector.
  - Functionals can be used in place of loops when there is no dependency between iteration
- **An Odyssey with purrr**
  - Type stable function output is preferred.

Acknowledgements

- Hadley Wickham's talk on "Managing many models with R" at Edinburgh *R* User Group
- Hadley Wickham's talk on "Expressing yourself with R"
- ADV-R Chapter 11: Functionals
- Brian Lee Yung's forthcoming book: Modeling Data With Functional Programming In R