Lecture 14: Jul 5, 2018

Functionals

- *Ubiquitousness of Functions*
- *Functionals*

James Balamuta
STAT 385 @ UIUC

Today's 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

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

When we talk about
high performance computing

On Today's Agenda

Functions are Objects

… pulling out function definitions …

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

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

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

Extracting Function Properties

... pulling out function definitions ...

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

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

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

Function Environment Scope

… use of variables within a function …

```
# 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
```

Local vs. Global Environments

… *R*'s scoping of values …

```
# Define value in global environment
# (e.g. outside of the function)
value = 3
multiple_constant = function(x) {
    # `value` is not been defined in the function.
    return(value * x)
}
multiple_constant(5)
```

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

Hidden Function Calls

Addition

Assignment

Subset

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

10 + 25
# [1] 35

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

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

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

```
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 …

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

do_operation = function(f, x, y) {
    f(x, y)
}

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

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

Your Turn

1. determine the function properties of **mean()**
2. spot the error in the function given below

```
# Note `value` has not been defined.
x = rnorm(10)
n = length(x)
my_func = function(x) {
    summed = 1/n * sum(x)
    summed
}
my_func(x)
```

Functionals

Specifying Missingness

… trying to set a missing value given a value …

```
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 and create a recipe …

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

# Apply behavior 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…

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

# Apply uniformly the action 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 ???

```
# Apply uniformly the behavior 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 uniformly the behavior 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 ???

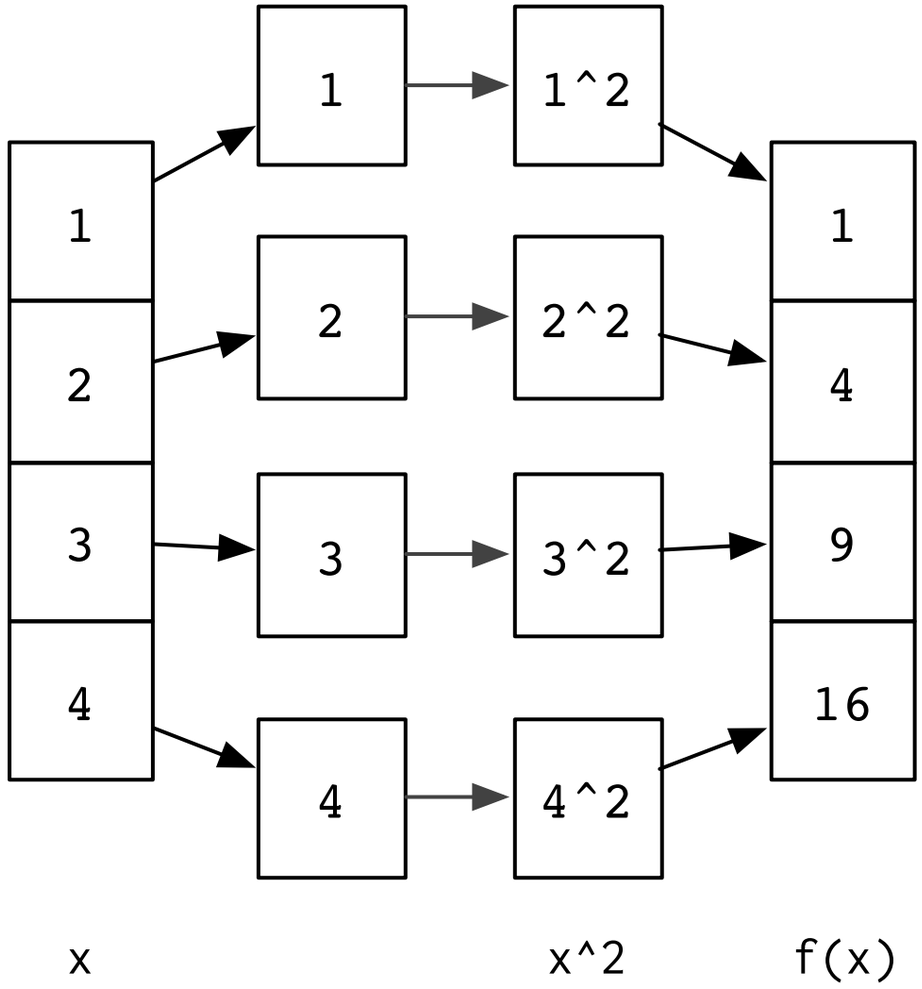
```
# Apply uniformly the behavior to columns
for(i in seq_len(ncol(my_df))) {
    my_df[, i]= code_missing(my_df[, i])
}
```

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?

Functional Programming

… three tenets …

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

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

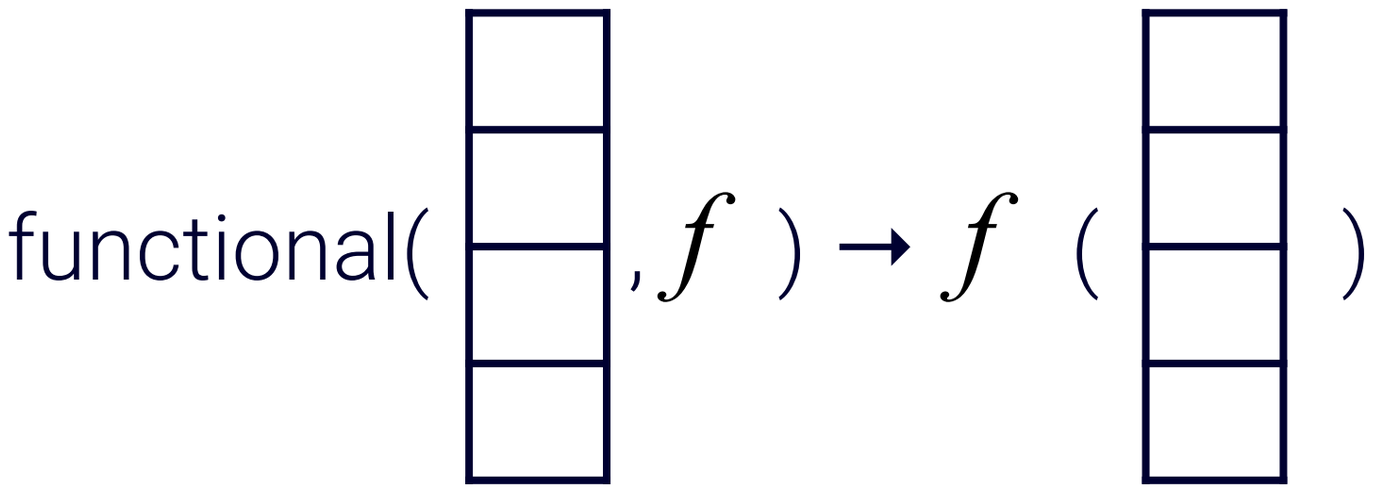
3. **Closures**
   e.g. functions returned with
   an external scope

Functions are the **lingua franca**

common language of all programming languages

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



Functionals in Practice

```r
call_func = function(x, f) {
  # call the function `f`
  # with data `x`
  f(x)
}

x = c(-2, 0.3, 1.2, 4.8)

call_func(x, mean)
# [1] 1.075

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

What if we have larger collections?

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	matrix
mapply	Apply a Function to Multiple List or Vector Arguments	vector, matrix, array, list

Functionals

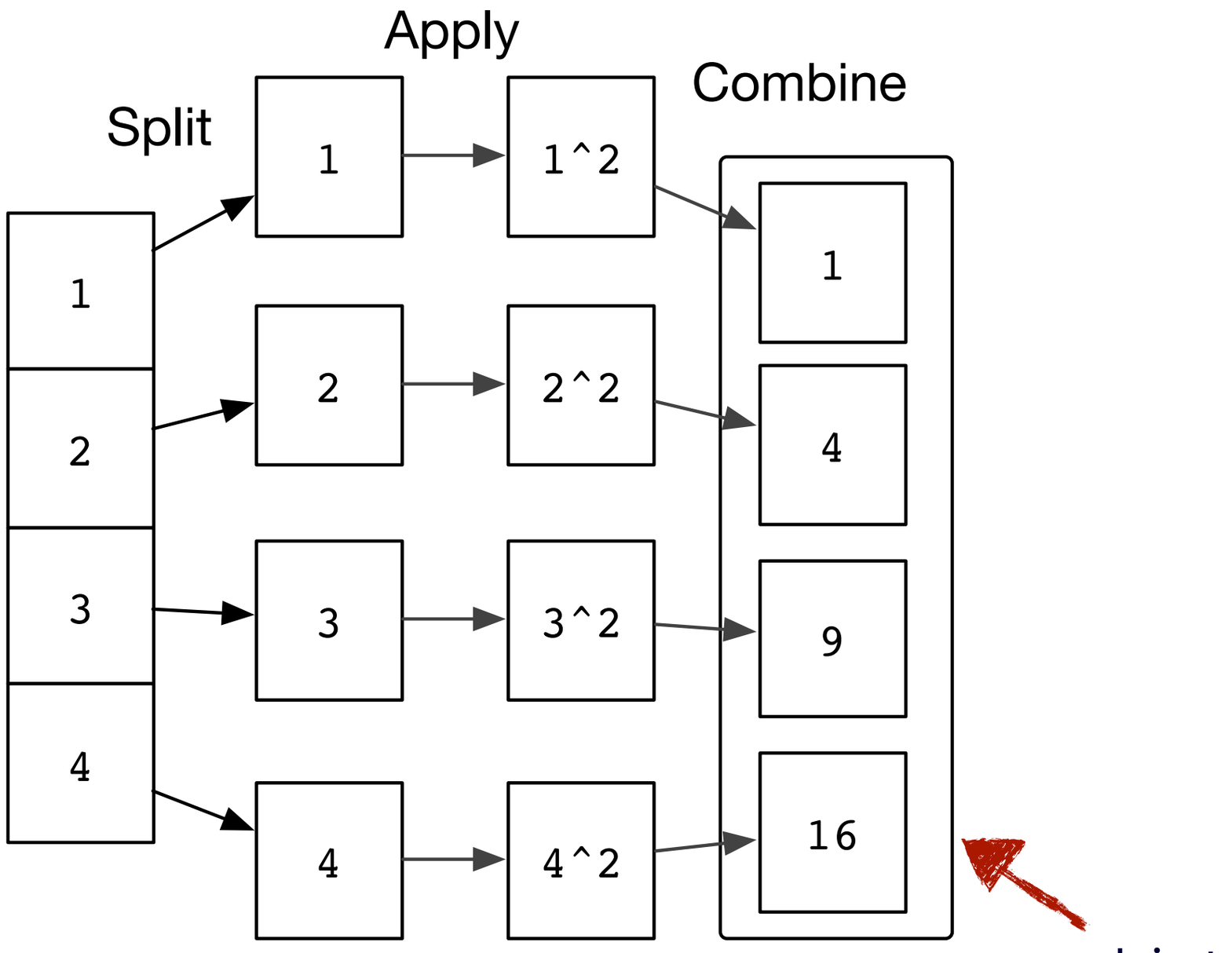
```r
x = c(-2, 0.3, 1.2, 4.8)

# 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 in a Loop Context

```
# Functional to mimic lapply()
my_lapply = function(x, f) {
  out = vector('list', length(x))

  for(i in seq_along(out)) {
    out[i] = f(x)
  }

  out
}

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

Breakdown of lapply

… lapply under Split-Apply-Combine …



Functionals Emphasize **Action**

… iteration vs. functional usage …

```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
```

Apply on Rows

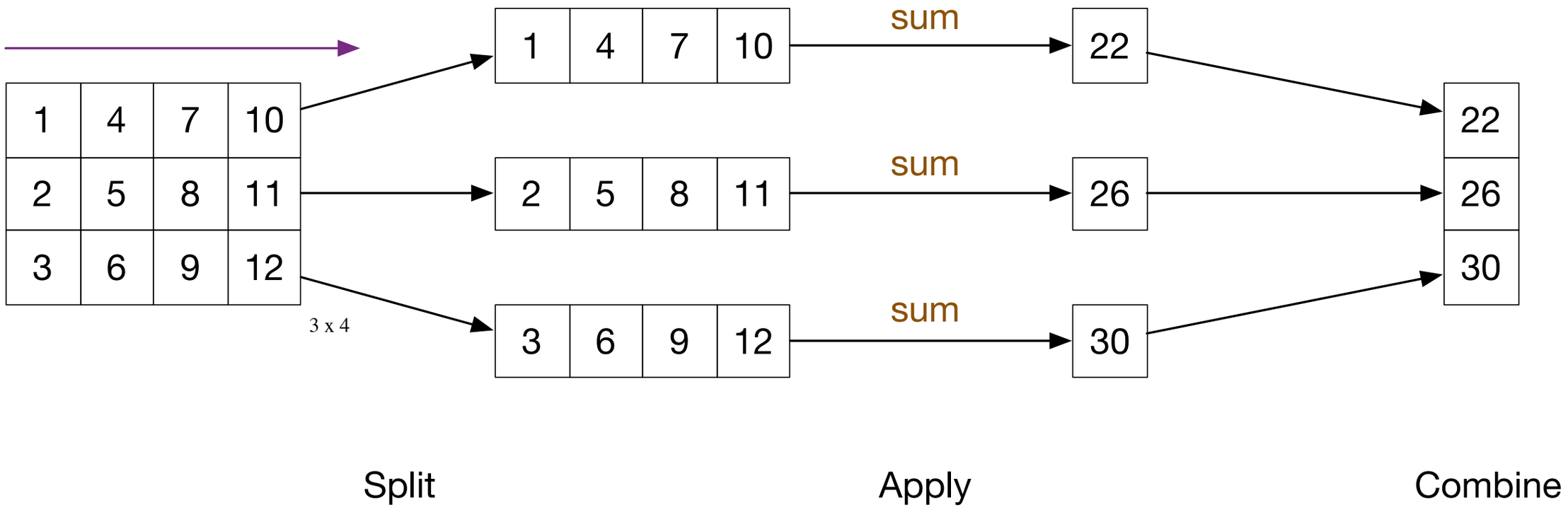
… applying a function to just rows …

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

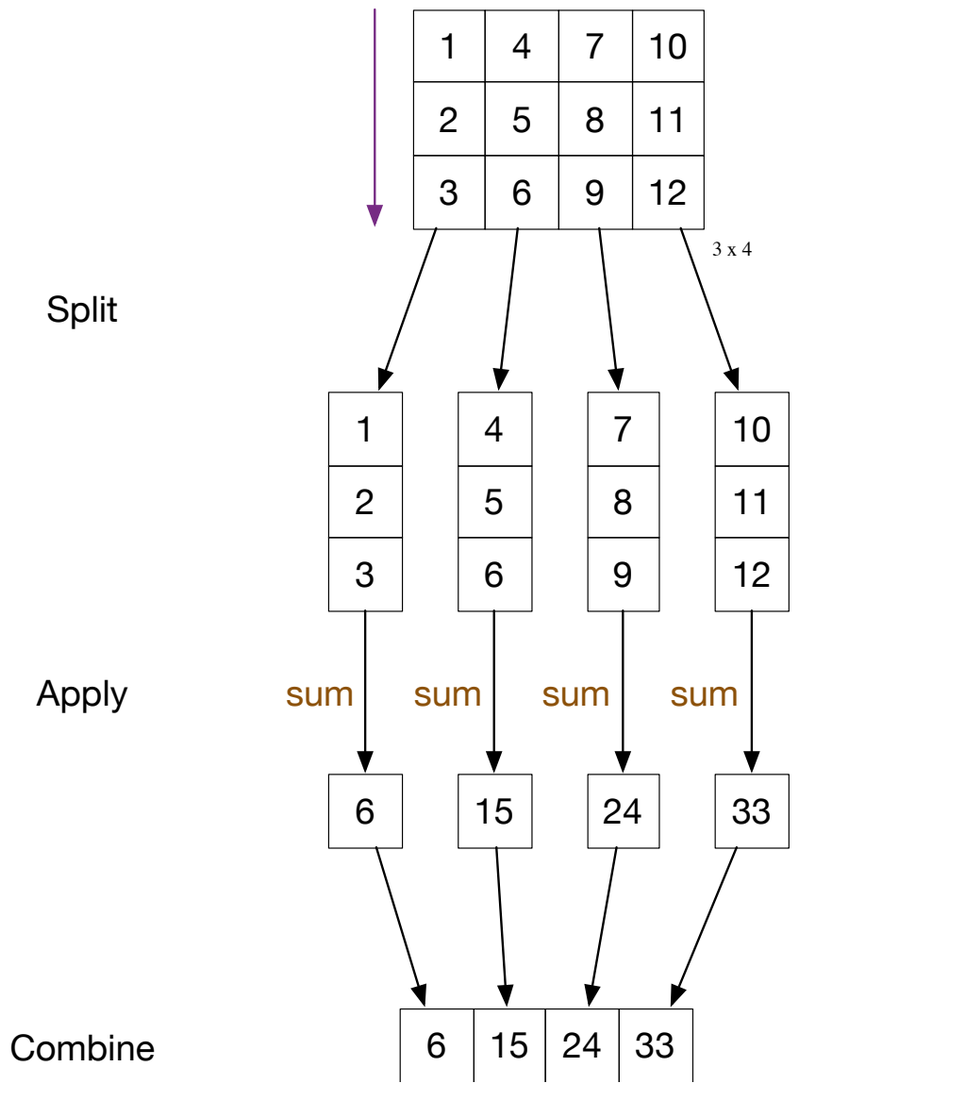
… apply with margin = 2 targets columns …

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

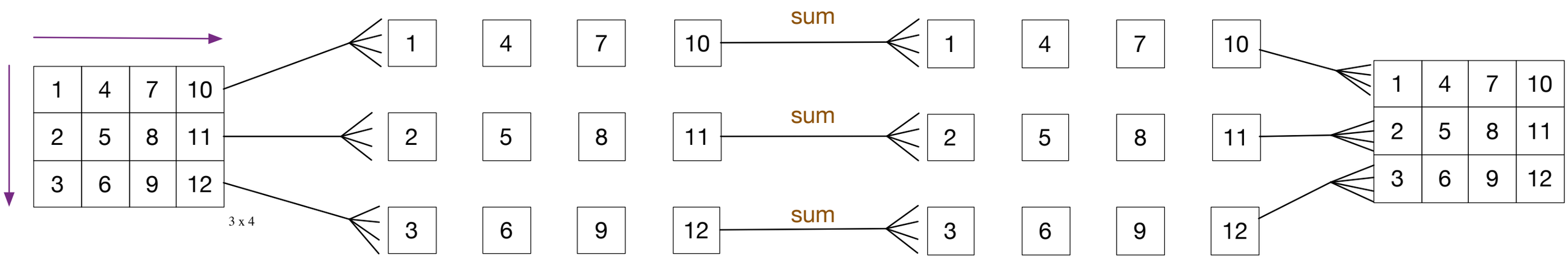
… apply over both rows and columns on a matrix …

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 …

```
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))

all.equal(version_one, version_two)
```

Power of Functionals

```
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 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
```

Your Turn

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

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

```
?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

1. Determine the classes of `mtcars`

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))
```

Recap

- **Background**
  - *R* is a functional language
  - Functions are objects
- **Functionals**
  - Functionals can be used in place of loops.
    - Assumes that there is no dependency between iteration

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