Functional Programming III: Functionals

UNIVERSITY OF
SAN FRANCISCO

James D. Wilson

BSDS 100 - Intro to Data Science with R

Functional Programming

- Assume you are given the following data frame

```
> myDataFrame_01
   A  B C   D   E F
1  1  6 1   5 -99 1
2 10  4 4 -99   9 3
3  7  9 5   4   1 4
4  2  9 3   8   6 8
5  1 10 5   9   8 6
6  6  2 1   3   8 5
```

- Your objective is to replace all of the `-99`s with `NA`s

Functional Programming

- You could—but shouldn't—iterate through each column manually, e.g.

```
myDataFrame_01$A[myDataFrame_01$A == -99] <- NA
myDataFrame_01$B[myDataFrame_01$B == -99] <- NA
...
myDataFrame_01$F[myDataFrame_01$F == -99] <- NA
```

Problems with Brute-Force Approaches

1. It's easy to make copy-paste mistakes
2. It makes bugs more likely
3. It makes updating code a HUGE pain in the arse
4. etc.

- Employ the Do Not Repeat Yourself (DRY) Principle

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [Thomas & Hunt, http://pragprog.com]

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

Functional Programming [EXAMPLE 1]

Let's write a function with the objective of replacing all `-99`s in a single column with `NA`s

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
}
```

- Will the code above work as intended? Hint: **no**. Why not?

Functional Programming [EXAMPLE 1]

The following **does** work as intended:

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}

myDataFrame_01$A <- fix99s_byCol(myDataFrame_01$A)
...
myDataFrame_01$F <- fix99s_byCol(myDataFrame_01$F)
```

- This reduces but doesn't eliminate the potential for errors
- There is no gain in efficiency (repetitive code is still required)

Functional Programming [EXAMPLE 1 REVISITED]

- What if there were a function that could iterate not only across all rows of a column checking for `NAs`, but also across all columns of a data frame?
- `lapply()`—from the generic family of `apply()` functionals—takes three inputs
  1. A list
  2. A function (applied to each element of the list)
  3. `...` (other arguments to pass to the function)

Functional Programming [EXAMPLE 1 REVISITED]

- `lapply()` applies the function to each element of a list a returns the new list

n.b. We can employ `lapply()` here because data frames are lists

Definition `lapply()` returns a list of the same length as (the list) X, each element of which is the result of applying a function to the corresponding element of X.

Functional Programming [EXAMPLE 1 REVISITED]

- `lapply()` applies the function to each element of a list a returns the new list

n.b. We can employ `lapply()` here because data frames are lists

Definition `lapply()` returns a list of the same length as (the list) X, each element of which is the result of applying a function to the corresponding element of X.

Functional Programming [EXAMPLE 1 REVISITED]

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}

> myDataFrame_01 <- lapply(myDataFrame_01, fix99s_byCol)

> str(myDataFrame_01)
List of 6
 $ A: num [1:6] 1 10 7 2 1 6
 $ B: num [1:6] 6 4 9 9 10 2
 $ C: num [1:6] 1 4 5 3 5 1
 $ D: num [1:6] 5 NA 4 8 9 3
 $ E: num [1:6] NA 9 1 6 8 8
 $ F: num [1:6] 1 3 4 8 6 5
```

- This almost worked...but not quite

Functional Programming [EXAMPLE 1 REVISITED]

Here are two ways to correct the previous function call so that it returns a data frame

1.
```
> myDataFrame_01 <-
as.data.frame(lapply(myDataFrame_01,
fix99s_byCol))
```
2.
```
> myDataFrame_01[] <- lapply(myDataFrame_01,
fix99s_byCol)
```

Functional Programming [EXAMPLE 1 REVISITED]

Employing functional programming, as in the previous example, has many advantages

1. It is very compact
2. If the code for a missing value changes, it only needs to be updated in a single location
3. It works for any number of columns, so you don't need to specify the number of columns, therefore avoiding potential mistakes
4. All columns are evaluated uniformly
5. You can generalize the technique to a subset of columns if preferred

```
> myDataFrame_01[1:3] <- lapply(myDataFrame_01[1:3], fix99s_byCol)
```

Adding Arguments

- What if different columns employed different coding schemes for missing values, e.g., `-99`, `-999` and `-8888888`?
- You could end up copy/pasting the function

```
fix99s_byCol <- function(myCol) {
  myCol[myCol == -99] <- NA
  myCol
}
```

and replacing the `-99` in `myCol[myCol == -99] <- NA`, with a updated values in each copy/paste so that you end up with three different functions (but this is not efficient)

Adding Arguments

We can simply add an argument to the previous code as follows

```
fixMising <- function(myCol, myValue) {
  myCol[myCol == myValue] <- NA
  myCol
}
```

Anonymous Functions

- The following code is equivalent and permissible

```
myFunc_26 <- function(myCol) {
  myCol <- myCol + 3
  myCol
}

lapply(myDataFrame_01, myFunc_26)

#########################

lapply(myDataFrame_01, function(x) x + 3)
```

- What you are observing in the lower half of the code in an anonymous function, i.e., a function that does not have a name

Lab

- Create a compact and robust function which, when passed an $n \times m$ numeric data frame, returns, for each column, the
  1. Mean
  2. Median
  3. Standard Deviation
  4. Variance
  5. Quantiles
  6. IQR
- NOT THE BEST SOLUTION (but it works)

```
mySummaryFunc <- function(myCols) {
  c(mean(myCols), median(myCols), sd(myCols), var(myCols),
    quantile(myCols), IQR(myCols))
}

lapply(myDataFrame_01, mySummaryFunc)
```

Functionals

A functional is a function that takes a function as an input and returns a vector as an output

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TR
```

This works as follows

1. We create a functional named `myFunctional_01`
2. We pass the argument `myFuncArg` to `myFunctional_01`, **where the argument is itself a function**
3. The argument `myFuncArg` is then called using the parameters defined in the inline, anonymous function.

Functionals

When we call the functional

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TR
```

we get the following results

```
> myFunctional_01(mean)
[1] 0.5194186

> myFunctional_01(mean)
[1] 0.5038302

> myFunctional_01(min)
[1] 0.000956069

> myFunctional_01(sd)
[1] 0.2846844
```

Why use Functionals?

- A common use of functionals is as an alternative to *for* loops
- *For* loops have a reputation for being slow in `R`
- The real advantage of using functionals is the ability to express a clear, specific objective in a single statement
- The probability of generating bugs in your code decreases

The `apply()` Family of Functionals

The `apply()` family of functionals are often used in lieu of *for* loops, coming in a variety of flavors (not exhaustive)

Functional	Input	Output
`apply()`	Array/Matrix	Vector/Array
`lapply()`	Vector/List	List
`sapply()`	Vector/List	List
`vapply()`	Vector/List	Vector

Let's examine a few examples to convince ourselves that the `apply()` family of functionals are truly useful

Lab

Using X

1. Write code that **does not contain** functionals that computes the mean of each column of data
2. Employ your functional of choice to write code that computes the mean of each column of data

Lab Solution

1. Write code that **does not contain** functionals that computes the mean of each column of data

```
> myColMeans_01 <- numeric(ncol(X))

for (i in 1:3) {
  myColMeans_01[i] <- mean(X[ , i])
}
```

2. Employ your functional of choice to write code that computes the mean of each column of data

```
> (myColMeans_02 <- apply(X, 2, mean))
```

The `apply()` function

- `apply()` coerces input to either a matrix (in 2 dimensions) or an array (in $>$ 2 dimensions), therefore the second argument indicates the dimension over which to apply the function
- The `apply()` function outputs a numeric vector

How `lapply()` Works

`lapply()` is a wrapper for a common loop pattern

- It creates a container for output
- Applies the function `f()` to each element of a list
- Fills the container with the results
- Returns a list
- Use `unlist()` to convert the list to a vector

Note `lapply()` is particularly useful for working with data frames as data frames are lists

`sapply()` and `vapply()`

- Both operate similarly to `lapply()`, taking similar inputs, but they differ on output
- `sapply()` will guess at what type of output it should generate
  - `sapply()` is good for interactive coding as it minimizes typing and the coder is able to observe and rectify and unexpected output types
  - **do not** bury an `sapply()` in a function where it can generate an odd and difficult to trace error

`sapply()` and `vapply()`

- `vapply()` requires an additional argument, specifying the output type
    - More verbose than `sapply()`, it always generates consistent output based on argument specification, gives more informative error messages, and never fails silently, and is therefore more appropriate for use inside functions

Final Notes on Functionals

- For multiple varying arguments, use `Map()`
- Leveraging the fact that each iterations of `apply()` functionals is isolated from all others, this lends themselves well to parallelisation using `mclapply()` and `mcMap()` from the `parallel` package
- May also want to read up on the `purrr` package

Lab

Using `state.x77`

1. Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?
2. Repeat with `sapply()`. What type of output is returned?
3. Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

Lab

Using `state.x77`

1. Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?

```
c <- state.x77
lapplyResult <- lapply(3:8, function(i) return(cor(s[ ,2], s[ ,i])))
str(lapplyResult)
```

2. Repeat with `sapply()`. What type of output is returned?

```
sapplyResult = sapply(3:8, function(i) return(cor(s[, 2], s[, i])))
str(sapplyResult)
```

3. Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

```
tapply(s[, "Area"], state.region, sum)
```

Part V: Efficiency of Code

How to Quantify Code Efficiency

- A precise way to measure the speed of small blocks of code is microbenchmarking
- `R` has a package called `microbenchmark` which provides a range of tools for evaluating code efficiency

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
    expr    min      lq      mean median      uq     max neval
 sqrt(x)  3.959  4.2420  6.507298  6.607  7.3975  64.095  1000
   x^0.5 24.730 26.7105 32.004310 28.484 35.3140 110.297  1000
```

- By default, `neval = 100`

Some Context on Code Efficiency

It is useful to think about how many times a function needs to run before it takes one second

Microbenchmark	Interpretation
1 ms	1,000 calls takes one second
1 μs	1,000,000 calls takes one second
1 ns	1,000,000,000 calls takes one second

Practical Interpretation

It takes roughly 800 ns to compute the square root of 100 numbers using `sqrt()`. That means that if you repeated that operation a million times, i.e., compute the square root on 10,000,000 numbers, it would take 0.8 seconds.

system.time()

Wrapping code in `system.time()` will also give you the system time required to process code, but

1. `microbenchmark()` is far more precise
2. `system.time()` only runs the block of code once, therefore you need to mnually wrap `system.time()` in a loop to generate meaningful statistics

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
    expr    min     lq      mean median      uq     max neval
 sqrt(x)  3.834  3.971  6.823777  4.213  7.7275 639.492  1000
   x^0.5 24.094 24.804 30.690782 26.232 31.9885 100.101  1000
>
> system.time(for (i in 1:1000) x^0.5) / 1000
   user  system elapsed
2.7e-05 1.0e-06 2.8e-05
```