Introduction to Functions, Part II

Rebecca C. Steorts, Duke University

STA 325, Supplemental Material

Agenda

- Multiple functions: Doing different things to the same object
- Sub-functions: Breaking up big jobs into small ones
- Example: Back to resource allocation

Why Functions?

- Data structures tie related values into one object
- Functions tie related commands into one object
- Both data structures and functions are easier to work with

Defining a function

```
function.name <- function(arguments){
  # computations on the arguments
  # some other code
  # return desired output
}
```

What should be a function?

- Things you're going to re-run, especially if they will be re-run with changes
- Chunks of code you keep highlighting and hitting return on
- Chunks of code that are small parts of bigger analyses
- Chunks that are very similar to other chunks

Why You Have to Write More Than One Function

Meta-problems:

- You've got more than one problem
- Your problem is too hard to solve in one step
- You keep solving the same problems

Meta-solutions:

- Write multiple functions, which rely on each other
- Split your problem, and write functions for the pieces
- Solve the recurring problems once, and re-use the solutions

Writing Multiple Related Functions

- In machine learning want to do lots of things with models/algorithms: estimate, predict, visualize, test, compare, simulate, uncertainty, etc.
- Write multiple functions to do these things
- We can then call the functions in order to meet our underlying goal (estimation, prediction, visualization, etc.)

Consistent Interfaces

- Functions for the same kind of object should use the same arguments, and presume the same structure
- Functions for the same kind of task should use the same arguments, and return the same sort of value

(to the extent possible)

Keep related things together

- Put all the related functions in a single file
- Source them together
- Use comments to note ***dependencies***

Rest of the module

In the rest of the module, we will do the following:

– review Tukey's method for outliers – review the three functions (from lab 2) needed for Tukey's method – look at an exercise at how all these functions work together on the rainfall dataset from your first homework assignment

Tukey's method

Identifying outliers in data is an important part of statistical analyses. One simple rule of thumb (due to John Tukey) for finding outliers is based on the quartiles of the data:

- the first quartile Q_1 is the value $\geq 1/4$ of the data,
- the second quartile Q_2 or the median is the value $\geq 1/2$ of the data,
- and the third quartile Q_3 is the value $\geq 3/4$ of the data.

The interquartile range, IQR, is $Q_3 - Q_1$.

Tukey's rule says that the outliers are values more than 1.5 times the interquartile range from the quartiles — either below $Q_1 - 1.5IQR$, or above $Q_3 + 1.5IQR$.

Testing for outliers

```r
# Input: data
# Output: quartiles (first, third, iqr)
quartiles <- function(x) {
  q1<-quantile(x,0.25,names=FALSE)  # suppress undesired percentile names
  q3<-quantile(x,0.75,names=FALSE)
  quartiles <- c(first=q1,third=q3,iqr=q3-q1)
  return(quartiles)
}

# Input: data
# Output: outliers according to Tukey's rule
tukey.outlier <- function(x) {
  quartiles <- quartiles(x)
  lower.limit <- quartiles[1]-1.5*quartiles[3]
  upper.limit <- quartiles[2]+1.5*quartiles[3]
  outliers <- ((x < lower.limit) | (x > upper.limit))
  return(outliers)
}

# Input: no arguments
# Output: TRUE if all tests passes,
# o/w error at first failed test
test.tukey.outlier <- function() {
  x <- c(2.2, 7.8, -4.4, 0.0, -1.2, 3.9, 4.9, 2.0, -5.7, -7.9, -4.9,  28.7,  4.9)
  x.pattern <- rep(FALSE,length(x)); x.pattern[12] <- TRUE
  stopifnot(all(tukey.outlier(x) == x.pattern))
  return(TRUE)
}
test.tukey.outlier()
```

```
## [1] TRUE
```

Back to rain

Let's load in the data set on rainfall from the first homework assignment.

```r
# read in rainfall data set
rain <- read.table("data/rnf6080.dat")
```

Task 1

The entries of `-999` represent missing observations, not hours of negative rainfall. Replace the negative numbers with `NA`.

Task 1

We simply replace all values with a -999 with an NA

```
rain[rain==-999] <- NA
```

Task 2

Run the 6th column of the cleaned data through your `tukey.outlier` function.What error message do you get? Where is the error happening? Why is it happening?

That is, run the following command in the console (after making sure to load in the functions).

```
rain <- read.table("data/rnf6080.dat")
tukey.outlier(rain[,6])
```

Task 2

we obtain the error `Error in quantile.default(x, 0.25) : missing values and NaN's not allowed if 'na.rm' is FALSE`

In order to understand what's happening, we can take a look at this in the console, to understand this further. This error makes it look like the problem might be happening when we call the built-in function `quantile`, in our `quartile` function. Using `traceback` confirms this:

```
5: stop("missing values and NaN's not allowed if 'na.rm' is
4: quantile.default(x, 0.25, names = FALSE)
3: quantile(x, 0.25, names = FALSE) at #2
2: quartiles(x) at #2
1: tukey.outlier(rain[, 6])
```

This makes it seem like `quantile` is refusing to process the data, because it contains `NA` values. Does it?

```
sum(is.na(rain[,6]))
```

Task 3

Write a test case, based on the `x` vector from lab, which shows how you would like your outlier-detector to handle `NA` values. Add it to your testing function.

Task 3

When we have NA values, we need to decide whether we are going to say that they are outliers (return `TRUE`), or say that missing values are not outliers (return `FALSE`), or refuse to say either way (return `NA`).

The last two make more sense. We'll consider these two cases.

Task 3 (Case 1)

Case 1: Suppose we want the `NA`s to NOT count as outliers. Our test case might look like the following:

```
# Initialize to data
x.with.nas <- x
# Create a vector which modifies data slightly by
# adding an NA in the middle, at position 7
x.with.nas[7] <- NA
stopifnot(all(tukey.outlier(x.with.nas)==x.pattern))
```

That is, we create a vector which modifies `x` slightly by adding an NA in the middle, at position 7. That wasn't the location of an outlier before, so if we don't want `NA`s to count as outliers we should still return false there. We just add these three lines to `test.tukey.outlier`.

Task 3 (Case 2)

Case 2: Suppose we want to return `NA`. Our test case might look like the following:

```
x.with.nas <- c(x,NA)
x.with.nas.pattern <- c(x.pattern,NA)
stopifnot(identical(tukey.outlier(x.with.nas),x.with.nas.pa
```

(Annoyingly, `NA==NA` evaluates to `NA`, while `identical(NA,NA)` is `TRUE`.)

Task 3

The current code for `tukey.outlier` will {*fail*} *this test case; in fact, running* *`test.tukey.outlier`* *at this stage should produce the same error message as in the previous question.*

To keep the solutions to a reasonable length, we'll just cover the option of making *`NA`s* *NOT outliers, rather than returning NA for them.*

Remark: We will modify the *`tukey.outlier`* *code in the next task section.*

Task 4

Write a test case, based on the `x` vector from lab, which shows how you would like your outlier-detector to handle `NA` values. Add it to your testing function.

Task 4

The `quantile` function has an option for ignoring NA values in the vector we give it, `na.rm`. So we re-define `quartile` to make use of this. Below, we re-define all the functions to incorporate a new test when faced with an NA value (missing is not an outlier).

Task 4

```r
# Re-defining test.tukey.outlier to incorporate a new test case
  # Return FALSE when faced with an NA value, i.e., missing is
  # not an outlier
test.tukey.outlier <- function() {
  x <- c(2.2, 7.8, -4.4, 0.0, -1.2, 3.9, 4.9, 2.0, -5.7, -7.9, -4.9,  28.7,  4.9)
  x.pattern <- rep(FALSE,length(x)); x.pattern[12] <- TRUE
  stopifnot(all(tukey.outlier(x) == x.pattern))
  stopifnot(all(tukey.outlier(-x) == tukey.outlier(x)))
  stopifnot(all(tukey.outlier(100*x) == tukey.outlier(x)))
  x.with.nas <- x
  x.with.nas[7] <- NA
  stopifnot(all(tukey.outlier(x.with.nas)==x.pattern))
  return(TRUE)
}

# Redefine quartile to handle NAs
quartiles <- function(x) {
  q1<-quantile(x,0.25,na.rm=TRUE,names=FALSE)
  q3<-quantile(x,0.75,na.rm=TRUE,names=FALSE)
  quartiles <- c(first=q1,third=q3,iqr=q3-q1)
  return(quartiles)
}

# Redefine tukey.outlier to output FALSE where the input value
# is NA i.e., missing values are never outliers
tukey.outlier <- function(x) {
  quartiles <- quartiles(x)
  lower.limit <- quartiles[1]-1.5*quartiles[3]
  upper.limit <- quartiles[2]+1.5*quartiles[3]
  outliers <- ((x < lower.limit) | (x > upper.limit))
  outliers[is.na(outliers)] <- FALSE
  return(outliers)
}
```

Task 4

```
summary(tukey.outlier(rain[,6]))
```

```
##    Mode   FALSE    TRUE
## logical    4873     197
```

```
test.tukey.outlier()
```

```
## [1] TRUE
```

Task 5

How many observations in the 6th column of the rainfall data are anomalies according to your improved `tukey.outlier`? How many are anomalies in the whole data set?

Task 5

In the 6th column, there are

```
sum(tukey.outlier(rain[,6]))
```

```
## [1] 197
```

total observations that are anomalies according to our improved `tukey.outlier`.

In the entire data set, there are

```
sum(tukey.outlier(rain))
```

```
## [1] 20196
```

total observations that are anomalies according to our improved `tukey.outlier`.

Remark: you may have noticed that the first three columns actually give the calendar date of the observations, and it doesn't make