R Programming

Shirley Liao

6/18/2017

The set up

Today we're going to delve a little more into the programming aspect of R coding, which will allow us to simplify our code, make it more efficient (take less time to run) and allow us to create completely original functions tailored to our data analysis needs.

By the end of this lesson, you should be able to conduct your own simulation study. A simulation study is the statistical equivalent of a mathematical model. In it, we randomly generate data from a pre-determined distribution or model for the purposes of:

1. Applying a method and evaluating its accuracy in estimating something about the data
2. Observing what happens to estimates as sample sizes become very small or very large
3. Observing if the estimates are valid even when assumptions are relaxed or violated ("robustness")
4. Comparing multiple methods in terms of accuracy, efficiency (how uncertain we are about the estimate) and computational burden

Simulation studies are often published as papers!

Review: Generating random data

1. Randomly sample 10 numbers from 1 to 100 without replacement, ensuring that numbers 1-20 have double the chance of being sampled as numbers 21-100 (HINT: there are two ways to do this).
2. Brad was told that his test score was in the 94th percentile of test scores in the nation, and that scores were distributed according to a normal distribution with an average of 79 and a standard deviation of 7. What was Brad's test score?
3. Simulate and store random draws size 10, 20, 50 and 100 all from a normal distribution with a mean of 3 and a standard deviation of 8. Have R estimate the mean and standard deviation of these samples using the mean() and sd() functions. What do you notice about the quality of estimation? Why?
4. Simulate tossing 30 coins with rbinom() and sample().
5. A rule of thumb is that 5% of a normal distribution lies outside an interval approximately 2 standard deviations above or below the mean. To what extent is this true? How long are the intervals (in terms of standard deviations) which exclude 10%, 1% or 0.5% of the normal distribution? BONUS: How would you relate the relationship between interval length and "coverage" of the normal distribution to the concept of confidence intervals?
6. Diane is a surgeon testing a new procedure. If the procedure has a 20% failure rate, she will stop using it. She has thus far operated on 10 patients and there have been no failures. What is the probability of this occurence if the procedure actually does have a 20% failure rate?
7. Amy plays an unusual lottery where every number chosen is assigned a different binomial distribution. There are 8 numbers, and the first number has a binomial distribution with $n = 10$, $p = 0.1$, the second number has a binomial distribution with $n = 20$, $p = 0.2$ and on until the 8th number has a binomial distribution with $n = 80$,$p = 0.8$. Amy's numbers this week are: 3 8 27 27 48 19 32 52. What is her probability of winning? BONUS: which numbers would give her the largest chance of winning and what is that chance?

Lesson 1: Loops

"Looping" is an iterative process which completes a task by repeating the same procedures (or series of procedures) multiple times, enabling us to solve complicated problems with simple code. Look at loops when you find yourself copy-pasting code to perform the same action 3 or more times. Loops can be categorized into two sub-categories: iterative and conditional.

Iterative loops ("for" loops)

Iterative loops "repeat" procedures a set number of times, and that number is pre-determined before the loop begins. The loop "keeps track" of what repitition it's currently performing by use of a counter.

```
f = 0
for(i in 1:10){ #for i as i changes from 1 to 10
  f = f+rnorm(1)
}
f
```

```
## [1] 2.089412
```

What did this loop do? How many times did we repeat the loop? What was our counter?

Frequently, the counter is used within the loop to reference different values in a data structure via index or to store outputted data as an index of an empty data structure. This requires us to declare an empty data structure before coding the loop and fill it as the loop iterates. For example, say that we wish to create a "Fibonacci" vector, where we begin with a vector such that $x_1 = 1$ and $x_2 = 1$ where x_i is the value of the vector at index i. Then from the third space on, $x_i = x_{i-1} + x_{i-2}$. We wish to end up with a vector length 20:

```
X = c(1,1,rep(NA,18)) #creates an empty vector except for the first two numbers

for(i in 3:20){ #we wish to start at position 3 in the vector
  X[i] = X[i-1] + X[i-2] #the values we retrieve from the vector X change with every i
}
```

You can also call a function or multiple functions inside of a loop:

```
m = 0

for(i in 1:100){
  r = rnorm(100)
  m = max(m,max(r))
}

m
```

```
## [1] 4.157383
```

Nested loops

Nested loops are useful for manipulating data in multidimentional arrays or for hierarchical problems. Beyond two nested loops, however, they become very ineffcient:

```
m = matrix(NA,nrow=10,ncol=10)

for(i in 1:10){
  for(j in 1:10){
    m[i,j] = mean(c(i+j,i*j))
  }
}

m
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5  10.5
##  [2,]  2.5  4.0  5.5  7.0  8.5 10.0 11.5 13.0 14.5  16.0
##  [3,]  3.5  5.5  7.5  9.5 11.5 13.5 15.5 17.5 19.5  21.5
##  [4,]  4.5  7.0  9.5 12.0 14.5 17.0 19.5 22.0 24.5  27.0
##  [5,]  5.5  8.5 11.5 14.5 17.5 20.5 23.5 26.5 29.5  32.5
##  [6,]  6.5 10.0 13.5 17.0 20.5 24.0 27.5 31.0 34.5  38.0
##  [7,]  7.5 11.5 15.5 19.5 23.5 27.5 31.5 35.5 39.5  43.5
##  [8,]  8.5 13.0 17.5 22.0 26.5 31.0 35.5 40.0 44.5  49.0
##  [9,]  9.5 14.5 19.5 24.5 29.5 34.5 39.5 44.5 49.5  54.5
## [10,] 10.5 16.0 21.5 27.0 32.5 38.0 43.5 49.0 54.5  60.0
```

Conditional loops ("while" loops)

Sometimes, you don't know how long you want the loop to iterate and instead wish it to stop when a certain condition has been met. Enter the "while" loop:

```
x = 0 #condition must be set outside of loop which allows loop to begin

while(x < 100){ #instead of an iterative statement, we have a conditional statement
  x = x + sample(seq(1,10),1)
}

x
```

```
## [1] 106
```

Be very careful when working with while loops that the condition which allows it to stop is achieveable by the variables manipulated inside the loop! If not, R will be stuck running an infinite loop and this could crash your computer (usually R is smart enough to send you an error message beforehand though).

Note: Reallocating memory is SLOW

Particularly tempting in while() loops (though also a problem with inexperienced coders using for() loops) is to call concatonation functions inside of the loop in order to build a data structure:

```
#print out all random draws from a standard normal until one of them is larger than 4
output = c()
x = 0

while(x<4 & x>-4){
  x = rnorm(1)
```

```
  output = c(output,x)
}
```

While this is okay for smaller iterations, the time it takes to recreate a new data structure and fill it with your data during each iteration really adds up! It's much faster to pre-create a data structure and fill it within the loop. If you do not know how large the data structure should be beforehand, you could always create one much larger than you suspect you need and cut it down later.

```
output = rep(NA,10000)
i = 0 #this is how you iterate in a while loop
x = 0

while(x<4 & x>-4){
  i = i + 1
  x = rnorm(1)
  output[i] = x
}

output = output[!is.na(output)]
```

Let's time them!

```
set.seed(1234) #ensuring they get the same string of random #s

#remaking the string at every iteration
output = c()
x = 0
system.time(
while(x<4 & x>-4){
  x = rnorm(1)
  output = c(output,x)
}
)
```

```
##    user  system elapsed
##   0.460   0.020   0.533
```

```
#filling the string
output = rep(NA,10000)
i = 0 #this is how you iterate in a while loop
x = 0

system.time(while(x<4 & x>-4){
  i = i + 1
  x = rnorm(1)
  output[i] = x
})
```

```
##    user  system elapsed
##   0.030   0.001   0.033
```

In general, avoid putting functions such as cat(), rbind(), cbind() or other concatonation procedures inside of loops.

Logical conditional statements

A while() loop is only one of several methods we can use to break out of a loop if some condition has been met. We glanced upon logical statements in a previous lecture, but now we will formalize the topic.

Logical statements include: - "<" lesser than/"<=" lesser or equal to - ">" greater than/">=" greater or equal to - "==" equal to - "!" is not (for example, "!=" means is not equal to)

You may also combine multiple logical statements using: - "&" and, used for piecewise/vector statements/"&&" used for wholistic statements - "|" or, used for piecewise/vector statements/"||" used for wholistic statements

Logical tests will output a boolean vector, indicating whether the tests have been passed.

```
3 >= 9
```

```
## [1] FALSE
```

```
x = c(2,3,5,3,5)
y = c(3,4,3,7,8)
x < 9
```

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

```
x < 9 & y > 3 #for which indicies is x smaller than 9 and y larger than 3
```

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

```
x < 9 | y > 3 #for which indicies is x smaller than 9 OR y larger than 3
```

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

```
x < 9 && y > 3 #is EVERYTHING in x smaller than 9 and y larger than 3
```

```
## [1] FALSE
```

Try the any() and all() functions with "x>4". What do these functions do?

You can create your own logical tests and there are many functions which do so for you! The only requirement is that the test output boolean data.

Conditional statements take in TRUE/FALSE statements and perform actions if the statement is TRUE. Conditional statements do NOT take in vectors of booleans.

```
x = 10

if(x<9){ #one conditional statement
  print("x is smaller than 9")
}

if(x<9){
  print("x is smaller than 9")
} else{
  print("x is not smaller than 9")
}
```

```
## [1] "x is not smaller than 9"
```

```
if(x<9){
  print("x is smaller than 9")
} else if(x<13){
  print("x is not smaller than 9 but is smaller than 13")
} else {
  print("x is not smaller than 13")
}
```

```
## [1] "x is not smaller than 9 but is smaller than 13"
```

Note that if()/elseif()/else() statements divide the event space into mutually exclusive areas. Only one of the conditional procedures will be performed. If you wish to perform two or more non-mutually exclusive tests you must use two if() statements instead of an if() and an elseif() statement. Also, order does matter!

```
x = 9

#makes sense
if(x > 9){
  print("x is larger than 9")
} else if(x>4){
  print("x is larger than 4")
}

#makes no sense, the second statement will never be used
if(x > 4){
  print("x is larger than 4")
} else if(x>9){
  print("x is larger than 9")
}
```

In loops, conditional statements are used to "divide up" your procedures. For example, we can use a loop to count up the number of even numbers in a vector:

```
count = 0
x = 1:1000

for(i in 1:100){
  if(x[i]%%2==0){ #if x[i] divided by 2 has a remainder of 0
    count = count+1
  }
}

count
```

```
## [1] 50
```

What if we want to count up the number of numbers divisible by 8 as well as the number of numbers divisible by 7?

```
count8 = 0
count7 = 0

for(i in 1:100){
  if(x[i]%%8==0){
    count8 = count8+1
  } else if(x[i]%%7==0){
```

```
    count7 = count7+1
  }
}
count8
```

```
## [1] 12
```

```
count7
```

```
## [1] 13
```

Conditional statements can also be used to "break" out of a loop, avoiding infinite looping or errors from compounding. Breaks can also serve the same function as while() loops:

```
#how many even numbers exist before the first number divisible by 7?
count = 0

for(i in 1:100){
  if(x[i]%%2==0){
    count = count+1
  } else if(x[i]%%7==0){
    break
  }
}
count
```

```
## [1] 3
```

Sometimes, instead of breaking we just want the loop to "pass over" this iteration without performing any later procedures. For example, if we wish to print all numbers 1 through 10 without printing 8:

```
for(i in 1:10){
  if(i==8){
    next
  }

  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 9
## [1] 10
```

Note: ifelse() versus else if()

While these two statements look similar, elseif() is a conditional statement which takes a boolean and MUST follow an if() statement while ifelse() is a free-standing function which replaces TRUE/FALSE outputs of logical tests with two outputs of your choosing.

```
x = c(3,4,5,6,32,65,7,43,23,6,23,1)
ifelse(x>10,"A","B")
```

```
##  [1] "B" "B" "B" "B" "A" "A" "B" "A" "A" "B" "A" "B"
```

Exercise 1:

1. Examine all the example loops in this section. For those which you are able, find an alternative, non-loop method to perform the same operation. For those which you cannot, explain why.
2. Write a while() loop which starts at 0 and prints odd numbers until it has printed 10 of them.
3. Write a nested for() loop (where the outer for loop increments a from 2 to 8 by 1, and the inner for() loop increments b from 1 to 6 by 2) that prints "a is less than b" if a<b.
4. Find the probability that, if you roll 3 fair dice, two of them will add up to the results of a third. (Hint: loop with multiple conditional statements)
5. Create an empty data frame. Inside a loop, add columns of length 20 to the data frame filled with numbers randomly generated from a standard normal distribution. Title the first column "A", the second column "B" and on until column "Z". Then, create another data frame with 26 columns, each appropriately titled. Iterate through the columns and fill them with random draws from the standard normal. Compare the computational time of these two procedures. Is adding a column to a data frame the same computational burdern that it is to concatonate two vectors or to add a row/column to a matrix?
6. Download the babyNames.csv dataset. Recode the "Sex" column (remember they had NAs for boy names and FALSEs for girl games) as a "boyNames" column with 1 for boy names and 0 for girl games using the ifelse() function.
7. BONUS PROBLEM: Imagine that you begin your walk on a coordinate system at a central point [0,0]. With every step you have equal chance of taking one step forward ([a+1,b]) or back ([a-1,b]), and equal chance of stepping left ([a,b+1]) or right ([a,b-1]). You will change location "vertically" AND "horizontally" every turn (ex: on your first turn there is an equal chance of you ending up in [1,1],[1,-1],[-1,1], or [-1,-1]). You end your walk if you go above 10 or below -10 in any direction. Perform this walk 100 times. How many steps, on average, are you able to take? Can you vectorize this problem?

Lesson 2: Vectorization

While loops are relatively straightforward, when working with large amounts of data they are extremely ineffcient. In some situations, you may be able to trade complexity for efficiency and perform the same operations via vectorization.

Vectorization refers to the application of functions, or a combination of functions to a vector/matrix of data at once, instead of performing that operation on one data point at once (loop). Vectorization doesn't necessarily involve "more code", but may be tricky because you must structure the problem in a different way.

Let's say that we wish to perform an imputation for missing data. Here we have a vector of data with some randomly missing data, and we wish to impute the mean of the dataset whenever there is an NA (this is called "single imputation").

```
z = rnorm(3000)
z[sample(1:3000,938,replace=FALSE)] = NA

zforloop = z
zforvector = z
```

Here is how we would do this via loop:

```
for(i in 1:3000){
  if(is.na(zforloop[i])){ #this is an example of a conditional statement, which we will talk about in t
    zforloop[i] = mean(zforloop,na.rm=TRUE)
  }
}
```

Here is how we would do this via vectorization:

```
zforvector[is.na(zforvector)] = mean(z,na.rm=TRUE)
```

As you can see, vectorization results in less code (and is faster), but requires more thoughtful consideration of the problem. A good rule of thumb for coding in R is to replace loops with vectorized arguments whenever possible, even if this means initially writing all code in loops for clarity, then going back through to find areas where you can speed things up.

Let's say that we wish to recreate our nested loop example (creating a matrix where each element in the matrix is a mean of the sum of its indices and the product of its indicies) using vectorization/no loops:

```
#matrix where element = row number
rowmatrix = matrix(rep(1:10,10),nrow=10,ncol=10)

#matrix where element = col number
colmatrix = t(rowmatrix)

newmatrix = ((rowmatrix+colmatrix)+(rowmatrix*colmatrix))/2
newmatrix
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  1.5  2.5  3.5  4.5  5.5  6.5  7.5  8.5  9.5  10.5
##  [2,]  2.5  4.0  5.5  7.0  8.5 10.0 11.5 13.0 14.5  16.0
##  [3,]  3.5  5.5  7.5  9.5 11.5 13.5 15.5 17.5 19.5  21.5
##  [4,]  4.5  7.0  9.5 12.0 14.5 17.0 19.5 22.0 24.5  27.0
##  [5,]  5.5  8.5 11.5 14.5 17.5 20.5 23.5 26.5 29.5  32.5
##  [6,]  6.5 10.0 13.5 17.0 20.5 24.0 27.5 31.0 34.5  38.0
##  [7,]  7.5 11.5 15.5 19.5 23.5 27.5 31.5 35.5 39.5  43.5
##  [8,]  8.5 13.0 17.5 22.0 26.5 31.0 35.5 40.0 44.5  49.0
##  [9,]  9.5 14.5 19.5 24.5 29.5 34.5 39.5 44.5 49.5  54.5
## [10,] 10.5 16.0 21.5 27.0 32.5 38.0 43.5 49.0 54.5  60.0
```

This "vectorization" of the problem uses more memory than a loop (why?) but this only becomes a problem if the matrices' size is very large, at which point the loop will be impossible to perform as well due to time constraints.

There are two main domains of operations where vectorization would not be a possible replacement for loops and those are:

1. Loops where each iteration is dependent on the results of previous iterations
2. Repeat use of functions that do not take vector arguments (rare for base R functions).

The latter of which can still be simplified with the use of apply() functions.

Apply functions (also called map functions)

Apply functions are not quite "vectorizations" as there is still a loop going on "in the background". In many cases, however, they will still perform faster than actual loops. The genral idea of an apply function is that it takes a function and "applies" is simultaneously to multiple inputs at once. There are several apply functions

out there, and many of them are repetitive with overlapping domains (as many functions in R are). In this class, we will introduce the best apply function to use for each data type:

1. Matrices: apply()

Takes in a matrix, outputs either a vector or matrix

```
x = matrix(rnorm(320),nrow=40,ncol=8)
apply(x,MARGIN = 2,FUN = mean)
```

```
## [1]  0.09650631 -0.07475451  0.24126578  0.04148674 -0.09295827 -0.19266642
## [7] -0.04155130  0.20229842
```

```
#MARGIN asks whether we wish to apply the function to the rows (1) or columns (2)
#FUN is the function name
apply(x,MARGIN = 2,FUN = quantile, probs = c(0.025,0.975))
```

```
##            [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## 2.5%  -2.212876 -1.401503 -1.140141 -1.635995 -1.646629 -1.574642
## 97.5%  1.855427  2.613588  1.931051  1.507539  1.505878  2.254594
##            [,7]      [,8]
## 2.5%  -1.649844 -1.603023
## 97.5%  1.227832  2.630133
```

```
#other function specifications should be made in the apply() function
```

2. Data frames: by()

Data frames may utilize apply() as well, but by() is more useful in certain circumstances because it allows you to choose a column which separates the data into several groups, upon which the chosen function will be applied. Say, for example, we wish to obtain a summary of our air quality data separated by month:

```
data("airquality")
by(airquality,airquality$Month,summary)
```

by() may be a bit tricky to use because it only works for functions which take data frames as inputs. For something like finding a mean, for example, it requires you to write a unique function which parses the subset of the data frame into your desired input:

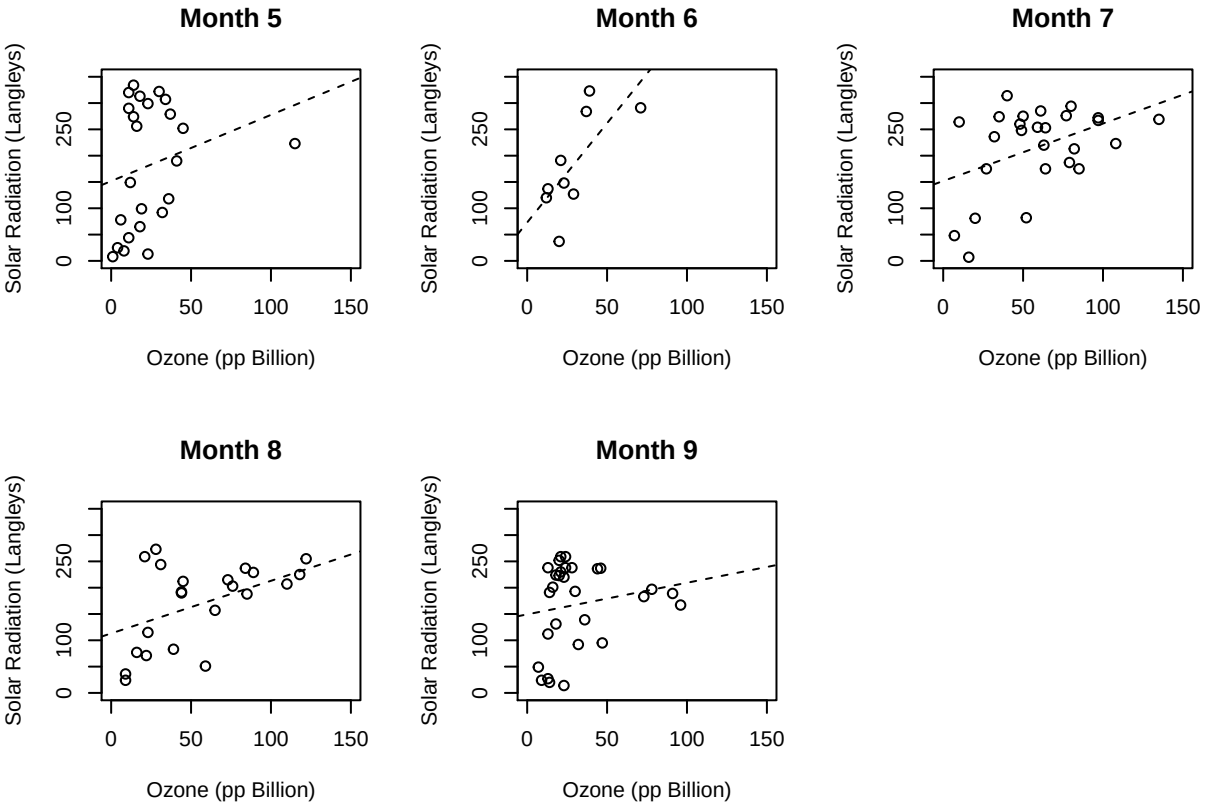
```
by(airquality,airquality$Month,function(x) {
  c(mean(x$Ozone,na.rm=TRUE),mean(x$Solar.R,na.rm=TRUE),mean(x$Wind,na.rm=TRUE),mean(x$Temp,na.rm=TRUE))
})
```

```
## airquality$Month: 5
## [1]  23.61538 181.29630  11.62258  65.54839
## ------------------------------------------------------------
## airquality$Month: 6
## [1]  29.44444 190.16667  10.26667  79.10000
## ------------------------------------------------------------
## airquality$Month: 7
## [1]  59.115385 216.483871   8.941935  83.903226
## ------------------------------------------------------------
## airquality$Month: 8
## [1]  59.961538 171.857143   8.793548  83.967742
## ------------------------------------------------------------
## airquality$Month: 9
## [1]  31.44828 167.43333  10.18000  76.90000
```

The power of the by() function is that you can coerce it to doing more complicated things, once you get the knack of writing your own functions (which we will learn in the next lesson):

```r
par(mfrow=c(2,3))
by(airquality,airquality$Month,function(x) {
 plot(x$Ozone, x$Solar.R, xlab="Ozone (pp Billion)", ylab="Solar Radiation (Langleys)",xlim=c(0,150),yl
  abline(lm(Solar.R ~ Ozone, x), lty=2)
  title(paste("Month",x$Month[1]))
})
```

```
## airquality$Month: 5
## NULL
## ------------------------------------------------------------
## airquality$Month: 6
## NULL
## ------------------------------------------------------------
## airquality$Month: 7
## NULL
## ------------------------------------------------------------
## airquality$Month: 8
## NULL
## ------------------------------------------------------------
## airquality$Month: 9
## NULL
```



Exercise 2:

1. Download the "airquality" dataset from base R and find the mean, standard deviation and 95th percentile of each column.
2. Write a loop where in every iteration (for 1000 iterations) one random number is drawn from a

Normal(0,1) distribution and one number drawn from a Gamma(2,5) distribution. Find the percentage of time when the number from the normal distribution is higher. Then perform this same operation without the use of loops.

3. Consider the vector x = c(3,4,5,3,2,5,6,7,3,3,5,4,2,2,2). Find the indicies of all non-unique numbers in this vector (all numbers which are repeated at least once) without a loop (HINT: look up the unique() function).

Lesson 3: Functions

So far in this class we have learned how to use base R functions as well as install packages with more functions. Now we will learn how to write our own. Writing functions is one way to automate your code, if you must perform the same set of procedures multiple times on multiple data sets. Learning not only how to code functions, but how to code them effectively is essential for reproduceable research and writing your own R package!

Functions consist of three parts: 1. Input 2. Mechanics 3. Output

And always uses the syntax:

```
functionname = function(input){
  mechanics

  output
}
```

Think of input as something you are feeding into a machine - carefully consider which data types and structures your function should be able to accomodate, as well as any specifications you wish the user to have control of (and what defaults you may wish to use instead).

```
#Here is a simple function which takes in an integer and returns the cube root of that number

cuberoot = function(x){
  x^(1/3)
}

cuberoot(27)
```

```
## [1] 3
```

```
#Maybe we wish the user to be able to choose which root they are interested in

root = function(x,rootnum){
  x^(1/rootnum)
}

root(27,3)
```

```
## [1] 3
```

```
#But we want to set a default anyway

root = function(x,rootnum=3){
  x^(1/rootnum)
}

root(27)
```

```
## [1] 3
```

The mechanics of your function may utilize loops, conditional statements and other functions.

```
#create a function which takes as an input a month, day and year
##as well as desired output type
#it then outputs the amount of output type (day/month/year)
##difference between the inputted date and today's date

datediff = function(month, day, year, type="day"){
  #retrieve today's date, turn it into a string, split the string into 3 numbers (characters),
  ##take the outputted list and call the first vector, then turn that vector into numbers
  today = as.numeric(strsplit(as.character(Sys.Date()),"-")[[1]])

  #find the difference of the two dates in days

  #difference in years
  yeardiff = (today[1]-year)*365

  #difference in months
  if(month %in% c(6,9,11,4)){
  monthdiff = (today[2] - month)*30
  } else if(month == 2){ #we will ignore leap years for now
  monthdiff = (today[2] - month)*28
  } else {
  monthdiff = (today[2] - month)*31}

  #difference in days
  daydiff = today[3] - day

  diff = yeardiff+monthdiff+daydiff

  if(type=="day"){
    diff
  } else if(type == "month"){
    diff/30.5
  } else if(type =="year"){
    diff/365
  } else {
    print("invalid type")
  }
}

datediff(9,21,2005,type="year")
```

```
## [1] 11.75068
```

The output of your function can be any data type or a printed message. Think about how this will change the functionality of your function.

```
x = sample(1:100,replace=TRUE)

#here is a function which counts the number of even numbers in your vector
##outputting an integer
counteven = function(x){
  count = 0
```

```
  for(i in 1:length(x)){
    if(x[i] %% 2 ==0){
      count = count+1
    }
  }
  count
}
counteven(x)
```

```
## [1] 50
```

```
#here is a function which returns a vector of all even numbers
returneven = function(x){
  evens = rep(NA,length(x))
  for(i in 1:length(x)){
    if(x[i] %% 2 ==0){
      evens[i] = x[i]
    }
  }

  evens[!is.na(evens)]
}
returneven(x)
```

```
##  [1]  20  94  40  80  10  16  42  38  94  10  56  78  26  54  28  88  60
## [18]  60  44  48  28  90  34  92  76 100  60  92  34  16  92   2  80   8
## [35]  36  44  52  86  56  44  88  58  56  96  86  32   4  90  16  50
```

```
#here is a function which returns a data frame where columns include
##the indicies of even numbers as well as the value

returneven2 = function(x){
  evens = data.frame(rep(NA,length(x)),rep(NA,length(x)))
  names(evens) = c("value","index")

  for(i in 1:length(x)){
    if(x[i] %% 2 ==0){
      evens[i,] = c(x[i],i)
    }
  }

  evens[!is.na(evens$value),]
}
head(returneven2(x))
```

```
##   value index
## 1    20     1
## 2    94     2
## 4    40     4
## 6    80     6
## 7    10     7
## 9    16     9
```

Note: Is a return() statement necessary?

Looking at the examples of functions above, it may be difficult to parse at first glance what the "output" is supposed to be. R reads the output of a function as the last unassigned expression. If you return() the last unassigned expression, it accomplishes the same thing:

```
#same x as above

thirdnumber = function(x){
  x[3]
}
thirdnumber(x)
```

```
## [1] 93
```

```
thirdnumber2 = function(x){
  return(x[3])
}
```

The nuance of the return statement, however, is that it immedately ends your function afterwards.

```
#same x vector as above
#write a fxn to return the index of the first 100 in x
#output a message otherwise

x100 = function(x){
  for(i in 1:length(x)){
    if(x[i]==100){
      i #the function does not stop after this is identified
    }
  }

  print("no 100s in sample") #will always be printed
}

x100(x)
```

```
## [1] "no 100s in sample"
```

```
x1002 = function(x){
  for(i in 1:length(x)){
    if(x[i]==100){
      return(i) #ends function and loop immediately
    }
  }

  print("no 100s in sample")
  return()
}

x1002(x)
```

```
## [1] 49
```

There are, of course, ways to break out of functions without the use of return() statements.

Exercise 3:

1. Does the cuberoot function work on vectors? Does it work on matrices? Why? Can you modify that function to throw up an error message if anything but a single number is inputted as x?
2. Create an R function that inputs a numerical vector and an integer, outputting the number of elements in the vector which are more than [integer] away from the mean of the vector.
3. Create a function that given a numeric vector X returns the digits 0 to 9 that are not in X. If X= c(0, 2, 4, 8) the function returns c(1, 3, 5, 6, 7, 9)
4. Create a function that given two strings (one word each), check if one is an anagram of another.
5. Create an R script that returns TRUE if the elements of a vector x are strictly increasing.
6. Create a function that given a data frame, and a number or character will return the data frame with the character or number changed to NA.

Longer exercises:

Roulette

1. Create a function called "roulette" which takes the input of a "bet" (plus any additional information related to the type of bet) as well as a "stake" (in $) and outputs how much money the user won. (FYI: The pockets of the roulette wheel are numbered from 0 to 36. In number ranges from 1 to 10 and 19 to 28, odd numbers are red and even are black. In ranges from 11 to 18 and 29 to 36, odd numbers are black and even are red. The 0 is a green space. Roulette is played by tossing a single ball on the wheel, and the ball is allowed to randomly land on a single numbered space. Bets include: "singles" (betting on a single number), "high" (betting that numbers 19+ are chosen), "low" (betting that numbers 1-18 are chosen), "reds"/"blacks" (betting that the number chosen will be the colored space), and "evens"/"odds" (betting that the chosen number will be even or odd, as chosen). Winning your "bet" will double your money, except in the case of "singles" where it pays out 35x your stake. Losing your bet pays out nothing).
2. Simulate placing 10 bets of $100 each on a single bet type of your choice. What was your net gain/loss?
3. Simulate beginning the game with $1000 and playing $100 at a time on a single bet type of your choice until there is no money left. How many turns were you able to play?
4. Instead of sticking to a single bet type the entire game, try choosing a random bet type each time. You may need another function to perform this.
5. Compare your two betting strategies by finding your average net gain/loss (using the procedure in 2) in each case over 100 repeated iterations (10 bets of $100, 100 times).

Random vs. adaptive treatment assignment

This is a simplified version of a simulation study I completed in 2015:

You are tasked with designing a clinical trial to evaluate the effects of drug X versus a control (sugar pill) on preventing allergic reactions to dust. All patients were exposed to a dusty environment for the next three hours. The outcome data collected is binary: 1 if the patient experienced an allergic reaction during their exposure time and 0 if they did not.

You managed to recruit 200 patients and intend to perform a T test of proportions (t.test()) after all patients are recruited and their outcomes collected. If the T test shows that drug X is significantly better at the alpha = 0.05 level at preventing allergic reactions (p_x lower than p_c) then the FDA will approve the drug.

With the sugar pill, patients have a 70% chance of experiencing an allergic reaction.

- Assign the patients with equal probability to each treatment. Simulate the case where drug X performs exactly the same as the control ("null hypothesis"). What is your Type 1 error?
- Assign patients with equal probability to each treatment. Simulate the case where drug X gives patients a 40% chance of experiencing an allergic reaction ("alternative hypothesis"). What is your power?

Instead of assigning patients with equal probability to each treatment, we can use instead a "play the winner" design. This is how it works:

1. Assign the first 50 patients randomly, with equal probability, and collect their results.
2. Recruit patient 51 and assign them to the arm that is performing better (EX: if 19 out of 28 in the control arm and 17 out of 22 in the treatment arm had allergic reactions so far we assign patient 51 to the control arm). Collect their outcome and recalculate probabilities of success.
3. Repeat with the rest of the patients up to 200.

- Simulate the type 1 error and power under such a design, with the specifications in problems 1 and 2. How does this design compare to randomization?

-. Find the average number of people assigned to each arm under the null and alternative hypotheses.

The "play the winner" design is not longer a randomized assignment, which does lead to some theoretical problems. Let's try a design which still retains random assignment while still encorporating information about treatment performance:

1. Assign the first 50 patients randomly, with equal probability, and collect their results.
2. Recruit patient 51 and randomize them to the treatment arm with a probability = to the relative "success" in the treatment arm observed thus far (EX: if 19/27 people in the control arm experienced a reaction and 17/22 in the treatment arm did, the relative probability of not experiencing a reaction ("success") in the treatment arm = (5/22)/(8/27 + 5/22) = 43.4%) Collect their outcome and recalculate probabilities of success.
3. Repeat with the rest of the patients up to 200.

- Simulate the type 1 error and power under such a design, with the specifications in problems 1 and 2. How does this design compare to randomization and "play the winner"?
- Find the average number of people assigned to each arm under the null and alternative hypotheses.

Sources & Further Reading

- https://www.r-bloggers.com/vectorization-in-r-why/
- https://stackoverflow.com/questions/3505701/r-grouping-functions-sapply-vs-lapply-vs-apply-vs-tapply-vs-by-vs-aggrega http://www.r-exercises.com/tag/loops/