R Fundamentals Part 4: for loops and functions

Guadalupe Tuñón, Evan Muzzall, Shinhye Choi, Rochelle Terman, Dillon Niederhut

December 6, 2016

Table of Contents

Part 4 Learning objectives 1

1. Introduction 2

2. For loops 3

2. For loops - Example 1 3

2. For loops - break and stopifnot 5

Challenge 1 6

2. For loops - if, else, and ifelse 6

Challenge 2 8

3. Functions 8

3. Functions/ default arguments 9

3. Functions - functions-within-functions 10

3. Functions - unit conversion example 11

Challenge 3 13

3. Functions - Day 3 statistical tests function 13

4. Monte Carlo simulations 20

4. Monte Carlo simulations - Step 1 21

4. Monte Carlo simulations - Step 2 21

4. Monte Carlo simulations - include a plotting function 21

4. Monte Carlo simulations - Step 3 24

Challenge 4: The Birthday Problem 27

Acknowledgements 30

# Part 4 Learning objectives

1. Introduction
2. For loops
3. Functions
4. Monte Carlo simulations

Load your sleep\_VIM.csv file from Part 2:

getwd()

## [1] "/Users/evanmuzzall/Desktop/R-Fundamentals-master"

sleep <- read.csv("sleep\_VIM.csv", header=TRUE, stringsAsFactors = FALSE)

# 1. Introduction

Commands such as mean() are actually functions. Functions in R are a set of statements that together can provide smart, reusable ways to conduct our research. Assign functions in a way similar to how you assign an object (with <-). Functions are generally meant to be mapped to data structures or are used to write other functions. Using functions might allow you to write your code more efficiently.

What do you think the nrow() function does? Type a function name without the parentheses to see its underlying functional structure. Let's try nrow(), which returns the number of rows in a data frame.

nrow

## function (x)   
## dim(x)[1L]  
## <bytecode: 0x7fdc752fc8b0>  
## <environment: namespace:base>

# Now we can give nrow() an argument  
nrow(sleep)

## [1] 62

However, before we get to functions, let us review for loops because many functions have for loops embedded in them.

Set working directory and load the animals data frame - we will return to this a little later:

getwd()

## [1] "/Users/evanmuzzall/Desktop/R-Fundamentals-master"

animals <- read.csv("animals.csv", header=TRUE, stringsAsFactors=FALSE)  
str(animals)

## 'data.frame': 20 obs. of 5 variables:  
## $ Type : chr "Dog" "Pig" "Dog" "Cat" ...  
## $ Healthy : logi FALSE TRUE TRUE FALSE FALSE TRUE ...  
## $ Weight : num 4.06 4.49 5.29 6.63 3.81 ...  
## $ Height : int 10 6 7 6 8 6 7 9 5 10 ...  
## $ Progress: num 1.512 0.39 -0.621 -2.215 1.125 ...

# 2. For loops

Before we introduce how functions work and learn how to write them, let us discuss how a for loop works in R. A for loop repeats a block of code a certain number of times until a certain condition is met (or is no longer satisfied), telling the code to stop and potentially print/return some sort of output.

The basic syntax looks like this:

syntax: for (variable in sequence) {statement}

All for loops are precluded by for so that R knows you want to iterate over a loop.

The variable in (variable in sequence) is generally denoted with an i, which stands for "iterator". However, i should be thought of as a placeholder and can be represented by other means (e.g., x, donut, n, etc.). sequence is some sequence of numbers telling R how many times you want to iterate the code.

"{statement}" refers to the code that you want run over the sequence at each iteration i. Notice that it is contained within curly braces { } - this defines the boundary of the statement in the for loop.

## 2. For loops - Example 1

Let's create a matrix of 2 to the power of i where i is 1 to 10. First we create a null vector called vec which will serve as a placeholder for the output of the loop.

?rep  
?"for"  
?"in"

vec <- c(rep(NA, 10))  
vec

## [1] NA NA NA NA NA NA NA NA NA NA

Now, let's create a matrix of 2 to the power of i where i is 1 to 10

for(i in 1:10){  
 vec[i] <- 2^(i)  
 }   
vec

## [1] 2 4 8 16 32 64 128 256 512 1024

class(vec)

## [1] "numeric"

This code tells R that we want to raise 2 to the power of i, where i is 1:10.

We can also use indexing to modify only some elements. We now have an object called vec with defined values. What if we want to replace the first 5 elements of vec with 3 to the power of i instead?

You can change the sequence to tell R just to overwrite the first five positions!

for(i in 1:5){  
 vec[i] <- 3^i  
 }   
vec

## [1] 3 9 27 81 243 64 128 256 512 1024

#This has changed only the first five entries of `vec`

You can also loop over character data. First, create a name vector:

animal.names <- c("Cat", "Dog", "Pig", "Elephant", "Giraffe")  
animal.names

## [1] "Cat" "Dog" "Pig" "Elephant" "Giraffe"

Then, we create a NULL vector of the same length like we did above:

animals.length <- rep(NA, length(animal.names))  
animals.length

## [1] NA NA NA NA NA

Now give the null vector names:

names(animals.length) <- animal.names  
animals.length #See how we are building this from scratch?

## Cat Dog Pig Elephant Giraffe   
## NA NA NA NA NA

Finally, perhaps you want to perform some operation across animals.length. For example, count the number of characters in each animal name like this:

?nchar

for(i in animal.names){  
 animals.length[i] <- nchar(i)  
}  
animals.length

## Cat Dog Pig Elephant Giraffe   
## 3 3 3 8 7

## 2. For loops - break and stopifnot

For long loops, many intermediate commands can help us "end the looping" once a condition is met, or when a condition is no longer satisfied. Two particular useful ones are break and stopifnot.

break ends the looping once a certain condition is met.

stopifnot ends the loop when a certain condition is NOT met.

Let's begin by writing a for loop that outputs the numbers of the sequence 1:100

for(i in 1:100){   
 print(i)   
 }

if statements are frequently used to specify code to be evaluated when the condition is held. Here we tell the code that when i is equal to 50, stop the code:

?break

We usually have data that we pass into a for loop. Let's create an object called my\_nums that contains the integer vector 1:100. Let's test the *value* (rather than the index):

my\_nums <- 40:70  
for(i in 1:length(my\_nums)){  
 print(my\_nums[i])  
 if(my\_nums[i] == 50) break   
}

## [1] 40  
## [1] 41  
## [1] 42  
## [1] 43  
## [1] 44  
## [1] 45  
## [1] 46  
## [1] 47  
## [1] 48  
## [1] 49  
## [1] 50

# Numbers up to and including 50 are printed!

Let's quickly change our my\_nums object to include one NA in position 55.

my\_nums[15] <- as.numeric(NA)  
my\_nums

## [1] 40 41 42 43 44 45 46 47 48 49 50 51 52 53 NA 55 56 57 58 59 60 61 62  
## [24] 63 64 65 66 67 68 69 70

stopifnot works similarly. Here we tell the code to stop if i reaches a missing value (NA):

?stopifnot  
  
for(i in 1:length(my\_nums)){  
 print(my\_nums[i])   
 stopifnot(!is.na(my\_nums[i]))  
}

# Challenge 1

1. Write a for loop that outputs something
2. Insert a break or stopifnot command to tell it when to stop

## 2. For loops - if, else, and ifelse

if and else statements are control structures that let you control how a code should be iterated within a single for loop. You saw if above and is handy when you want to assign different tasks to different subsets of data using a single for loop.

if something happens, do "this" if something else happens, do "that"!

syntax: if (condition) {statement} else {other statement}

?"if"

x <- 7  
if(x > 7){  
 print(x)  
}else{ #`else` should not start its own line. Always let it be preceded by a closing brace on the same line.  
 print("NOT BIG ENOUGH!!")  
}

## [1] "NOT BIG ENOUGH!!"

Reassign x <- 10 here - what happens? Does indentation matter? Let me show you how I like to format my code (which is considered hard to read by most).

This also works in a loop. Here, we get all outputs for the loop:

x <- 1:10  
for (i in 1:10){  
 if(x[i] > 7){  
 print(x[i])  
 }else{   
 print("NOT BIG ENOUGH!!")  
 }  
}

## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] "NOT BIG ENOUGH!!"  
## [1] 8  
## [1] 9  
## [1] 10

#Super cool! :)

The ifelse function can be handy to recode data as long as you have two conditions that are binary/mutually exclusive.

syntax: ifelse(test, yes, no)

Let us begin by generating 10 random draws of two common housepets "cat" and "dog":

?"ifelse"

set.seed(1)  
animal <- sample(c("cat", "dog"), 10, replace=TRUE)  
animal

## [1] "cat" "cat" "dog" "dog" "cat" "dog" "dog" "dog" "dog" "cat"

Now we can recode the character data to numeric using ifelse():

animal <- ifelse(animal=="cat", 1, 0)   
animal

## [1] 1 1 0 0 1 0 0 0 0 1

... or recode the numeric data to some other character type:

animal <- ifelse(animal==1, "meow!?", "WOOF")  
animal

## [1] "meow!?" "meow!?" "WOOF" "WOOF" "meow!?" "WOOF" "WOOF"   
## [8] "WOOF" "WOOF" "meow!?"

... and recode the new character data type to logical type:

animal <- ifelse(animal=="meow!?", TRUE, FALSE)  
animal

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

# Challenge 2

1. Write a for loop that uses if
2. Write a for loop that uses if and else
3. Create and then recode some binary data using ifelse

# 3. Functions

In R, a function is a set of code that might be more useful if it is self-contained and/or is to be used repeatedly. For example, if you want to run the same satatistical tests on several datasets, you might want to write a function that contains instructions for all the tests once, so that you do not have to rewrite the code each time.

Every function has four parts and its basic syntax looks like this:

function.name <- function(x){ body of function }

1. function.name - like objects in R, it is helpful if you give your function a relevant name.
2. function(x) - function lets R know you are writing a function and (x) contains the inputs/arguments.
3. {body of function} - the body of the function is contained within curly braces { } and denotes the statements that you want R to evaluate.
4. The environment (global environment) that the function operates within.

For example:

?"function"

test\_function <- function(x){  
 x + 1  
}  
  
class(test\_function) # Returns the class of `test\_function`

## [1] "function"

formals(test\_function) # Shows the defined arguments

## $x

body(test\_function) # Displays the statements to be evaluated

## {  
## x + 1  
## }

environment(test\_function) # Returns the "global" environment

## <environment: R\_GlobalEnv>

test\_function # Shows your function as you have written it

## function(x){  
## x + 1  
## }

See how our function has R\_GlobalEnv as it's environment? That's because we defined it in the global environment. This means that if you tell a function to look for an object, it will look in the global namespace - not within the function itself.

To evaluate this function, call the name of the function with your argument x within parantheses. R then evaluates the body of the function and returns the desired output for the specified input.

If we want to see what x + 1 is equal to when x = 2, we type:

test\_function(2)

## [1] 3

## 3. Functions/ default arguments

We can also add a predetermined value for one or more arguments, which serves as a default value that we can change in particular applications. You can change the value of "y" - it is an optional argument because it is named and it has a default value. By convenction, required arguments do not have default values and are listed first in the function definition.

f <- function(x, y=2){  
 (x + y) / y  
}  
f(5)

## [1] 3.5

f(4, 4) # here, we do not need to specify `x` and `y` because R knows that the second argument is "y"

## [1] 2

f(4, y=4) # this is the same as f(4, 4)

## [1] 2

#does f(y=4, 4) work? Yes, but it is not recommended.  
f(y=4, 4)

## [1] 2

Let's try one more example where we define object z within the function.

f <- function(x, y=2){  
 z <- 5  
 (x + y) / z  
}  
f(4, 4)

## [1] 1.6

ls()

## [1] "animal" "animal.names" "animals" "animals.length"  
## [5] "f" "i" "my\_nums" "sleep"   
## [9] "test\_function" "vec" "x"

Now we ask: does z appear in your global environment? Why not? z is protected from what is happening outside the function!

Finally, let's think about this a little differently. Now let's define z in the global environment and see if the function can access it:

z <- 5 # define z outside of the function environment  
  
f <- function(x, y=2){  
 (x + y) / z  
}  
f(4, 4)

## [1] 1.6

ls()

## [1] "animal" "animal.names" "animals" "animals.length"  
## [5] "f" "i" "my\_nums" "sleep"   
## [9] "test\_function" "vec" "x" "z"

z is now defined in the global environment and the function can pull objects from the global namespace. For example, using constants that never change, define them in the global namespace. If it is unique to the function however, keep it in the function! Scope is tricky!

## 3. Functions - functions-within-functions

It is also common for functions to be declared within another function.

f <- function(x) {  
 y <- 1  
 g <- function (x) {  
 x + y  
 }  
 g(x)  
}  
f(1)

## [1] 2

What is going on here? Does g show up in your global environment? Why not? (hint: because it was defined in the functional environment instead of the global environment!)

This is important because it means that functions can be separated from the state of your computer (which is what makes them easy to parallelize). What we really mean here is that anything created inside the function environment doesn't show up in the global environment.

side note - R automatically returns the value of the last expression, so there is no need for an explicit return statement unless you want to break the function early

For example:

f <- function(x) {  
 if (x>5) {return("ERROR")}  
 y <- 4  
 g <- function (x) {  
 x + y  
 }  
 g(x)  
}  
f(x=1)

## [1] 5

f(x=10)

## [1] "ERROR"

## 3. Functions - unit conversion example

Unit conversion is a common obstacle in research. Let us write a function that converts inches to centimeters. For now however, pretend that we think that one inch is equal to 2.5 centimeters - we will show you how to update it below.

Define a function called in\_to\_cm and then enter Evan's height (74 inches):

in\_to\_cm <- function(x){  
 x \* 2.5  
}  
in\_to\_cm(74)

## [1] 185

#Evan is 185 cm tall (incorrectly assuming that 1 inch = 2.5 cm)

What if we want to know how tall I am in meters?

You could type:

function(x){  
 x \* 2.5 / 100  
}

...but this would be repeating yourself!

Then, when you figure out that the conversion factor is really *2.54*, not 2.5, you might update one function and forget to update the other - these inconsistencies can cause problems.

Instead, let's define a new function called in\_to\_m so that the output of in\_to\_cm is used in the new function!

in\_to\_m <- function(x){  
 in\_to\_cm(x) / 100  
}  
in\_to\_m(74)

## [1] 1.85

Now, if we go back and update in\_to\_cm, those changes automatically get propogated to in\_to\_m and we do not have to worry about updating it!

in\_to\_cm <- function(x){  
 x \* 2.54  
}  
in\_to\_m(74)

## [1] 1.8796

#Evan is actually 1.8796 meters tall

R is a bit quirky in that there is no such thing as an uncontained value, e.g. the number 74 is really a vector with length of one, and a value of 74 in position 1

74 == c(74)

## [1] TRUE

This means that R automatically broadcasts functions across vectors of any length. Notice that there is only one argument ("height.vec") even though it contains four different heights.

height.vec <- c(74,64,73,82)  
in\_to\_m(height.vec)

## [1] 1.8796 1.6256 1.8542 2.0828

Note this doesn't work with lists:

height.list <- list(74,64,73,82)

in\_to\_m(height.list)

# Challenge 3

Write a function called height\_metric that takes in height in inches and return\_meters = TRUE or FALSE and returns either the output of in\_to\_cm or in\_to\_m

height\_metric <- function(height\_inches, return\_meters=TRUE){  
 if(return\_meters==TRUE) {  
 in\_to\_cm(height\_inches)  
 #if return\_meters is TRUE do that  
 #else return cm  
 } else {  
 in\_to\_m(height\_inches)  
 }  
}  
height\_metric(height.vec,TRUE)

## [1] 187.96 162.56 185.42 208.28

Instead, a functional such as lapply() might be used to do so. Remember that a functional is a function that takes functions as arguments.

lapply(height.list, in\_to\_m)

## [[1]]  
## [1] 1.8796  
##   
## [[2]]  
## [1] 1.6256  
##   
## [[3]]  
## [1] 1.8542  
##   
## [[4]]  
## [1] 2.0828

Although the "dplyr" R package is useful for quickly applying functions to a matrix, there exists a whole family of functionals for this sort of thing. Learn more about the "apply" functional family [here](https://www.r-bloggers.com/using-the-apply-family-of-functions-in-r/), [here](https://www.r-bloggers.com/using-apply-sapply-lapply-in-r/), and [here](http://adv-r.had.co.nz/Functionals.html)

## 3. Functions - Day 3 statistical tests function

Remember back to Day 3 when you learned a t-test, ANOVA, cor.test, and lm? We can block these into a function quite conveniently, and we can even include plots!

First, refer back to your animals data frame.

str(animals)

## 'data.frame': 20 obs. of 5 variables:  
## $ Type : chr "Dog" "Pig" "Dog" "Cat" ...  
## $ Healthy : logi FALSE TRUE TRUE FALSE FALSE TRUE ...  
## $ Weight : num 4.06 4.49 5.29 6.63 3.81 ...  
## $ Height : int 10 6 7 6 8 6 7 9 5 10 ...  
## $ Progress: num 1.512 0.39 -0.621 -2.215 1.125 ...

head(animals)

## Type Healthy Weight Height Progress  
## 1 Dog FALSE 4.062035 10 1.51178117  
## 2 Pig TRUE 4.488496 6 0.38984324  
## 3 Dog TRUE 5.291413 7 -0.62124058  
## 4 Cat FALSE 6.632831 6 -2.21469989  
## 5 Pig FALSE 3.806728 8 1.12493092  
## 6 Cat TRUE 6.593559 6 -0.04493361

Then write your function to include ANOVA, Pearson correlation, and linear model information (as well as a boxplot). Notice that we are defining four objects within the function itself:

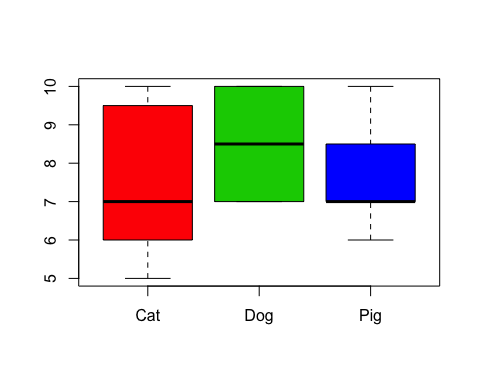
stats\_tests <- function(x){  
 sum\_aov <- summary(aov(Height ~ Type, data=animals))  
 Pearson <- cor.test(animals$Height, animals$Weight)  
 sum\_lm <- summary(lin.mod <- lm(Height ~ Weight, data=animals))  
 box <- boxplot(Height ~ Type, data=animals, col=c(2,3,4))  
stats\_list <- list(sum\_aov, Pearson, sum\_lm, box)  
names(stats\_list)[1:4] <- c("ANOVA summary info", "Pearson correlation info", "Linear model summary info", "Boxplot info")  
list(stats\_list)  
}

All we have to do is call the function on the argument x

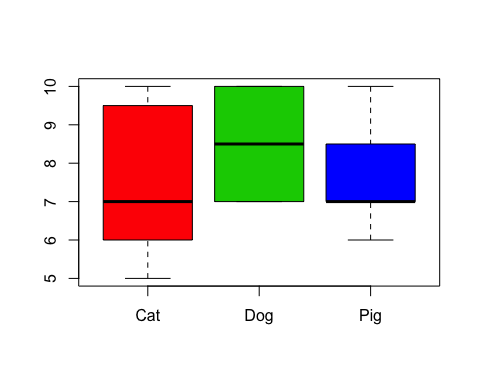
stats\_tests(x)

## [[1]]  
## [[1]]$`ANOVA summary info`  
## Df Sum Sq Mean Sq F value Pr(>F)  
## Type 2 3.16 1.579 0.53 0.598  
## Residuals 17 50.64 2.979   
##   
## [[1]]$`Pearson correlation info`  
##   
## Pearson's product-moment correlation  
##   
## data: animals$Height and animals$Weight  
## t = -1.8502, df = 18, p-value = 0.08076  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.71566804 0.05196763  
## sample estimates:  
## cor   
## -0.399746   
##   
##   
## [[1]]$`Linear model summary info`  
##   
## Call:  
## lm(formula = Height ~ Weight, data = animals)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -2.7262 -1.0758 -0.1898 1.2445 2.6226   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 10.9685 1.6958 6.468 4.39e-06 \*\*\*  
## Weight -0.5878 0.3177 -1.850 0.0808 .   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 1.585 on 18 degrees of freedom  
## Multiple R-squared: 0.1598, Adjusted R-squared: 0.1131   
## F-statistic: 3.423 on 1 and 18 DF, p-value: 0.08076  
##   
##   
## [[1]]$`Boxplot info`  
## [[1]]$`Boxplot info`$stats  
## [,1] [,2] [,3]  
## [1,] 5.0 7.0 6.0  
## [2,] 6.0 7.0 7.0  
## [3,] 7.0 8.5 7.0  
## [4,] 9.5 10.0 8.5  
## [5,] 10.0 10.0 10.0  
## attr(,"class")  
## Cat   
## "integer"   
##   
## [[1]]$`Boxplot info`$n  
## [1] 7 6 7  
##   
## [[1]]$`Boxplot info`$conf  
## [,1] [,2] [,3]  
## [1,] 4.909856 6.564903 6.104224  
## [2,] 9.090144 10.435097 7.895776  
##   
## [[1]]$`Boxplot info`$out  
## numeric(0)  
##   
## [[1]]$`Boxplot info`$group  
## numeric(0)  
##   
## [[1]]$`Boxplot info`$names  
## [1] "Cat" "Dog" "Pig"

info <- stats\_tests(x)

 We can subset the output like you learned in Part 2. For example, to return only the results of the Pearson correlation:

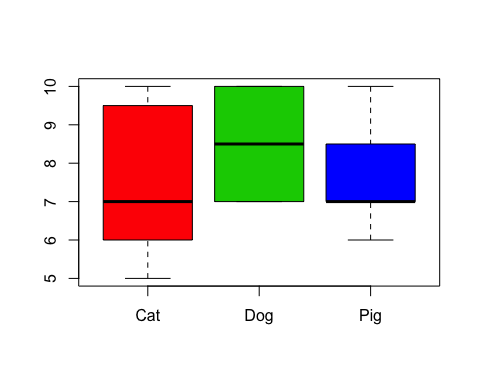
stats\_tests(x)[[1]][2]



## $`Pearson correlation info`  
##   
## Pearson's product-moment correlation  
##   
## data: animals$Height and animals$Weight  
## t = -1.8502, df = 18, p-value = 0.08076  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.71566804 0.05196763  
## sample estimates:  
## cor   
## -0.399746

Or return only the results of the linear model:

stats\_tests(x)[[1]][3]



## $`Linear model summary info`  
##   
## Call:  
## lm(formula = Height ~ Weight, data = animals)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -2.7262 -1.0758 -0.1898 1.2445 2.6226   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 10.9685 1.6958 6.468 4.39e-06 \*\*\*  
## Weight -0.5878 0.3177 -1.850 0.0808 .   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 1.585 on 18 degrees of freedom  
## Multiple R-squared: 0.1598, Adjusted R-squared: 0.1131   
## F-statistic: 3.423 on 1 and 18 DF, p-value: 0.08076

Or, call it like

info

## [[1]]  
## [[1]]$`ANOVA summary info`  
## Df Sum Sq Mean Sq F value Pr(>F)  
## Type 2 3.16 1.579 0.53 0.598  
## Residuals 17 50.64 2.979   
##   
## [[1]]$`Pearson correlation info`  
##   
## Pearson's product-moment correlation  
##   
## data: animals$Height and animals$Weight  
## t = -1.8502, df = 18, p-value = 0.08076  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## -0.71566804 0.05196763  
## sample estimates:  
## cor   
## -0.399746   
##   
##   
## [[1]]$`Linear model summary info`  
##   
## Call:  
## lm(formula = Height ~ Weight, data = animals)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -2.7262 -1.0758 -0.1898 1.2445 2.6226   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 10.9685 1.6958 6.468 4.39e-06 \*\*\*  
## Weight -0.5878 0.3177 -1.850 0.0808 .   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 1.585 on 18 degrees of freedom  
## Multiple R-squared: 0.1598, Adjusted R-squared: 0.1131   
## F-statistic: 3.423 on 1 and 18 DF, p-value: 0.08076  
##   
##   
## [[1]]$`Boxplot info`  
## [[1]]$`Boxplot info`$stats  
## [,1] [,2] [,3]  
## [1,] 5.0 7.0 6.0  
## [2,] 6.0 7.0 7.0  
## [3,] 7.0 8.5 7.0  
## [4,] 9.5 10.0 8.5  
## [5,] 10.0 10.0 10.0  
## attr(,"class")  
## Cat   
## "integer"   
##   
## [[1]]$`Boxplot info`$n  
## [1] 7 6 7  
##   
## [[1]]$`Boxplot info`$conf  
## [,1] [,2] [,3]  
## [1,] 4.909856 6.564903 6.104224  
## [2,] 9.090144 10.435097 7.895776  
##   
## [[1]]$`Boxplot info`$out  
## numeric(0)  
##   
## [[1]]$`Boxplot info`$group  
## numeric(0)  
##   
## [[1]]$`Boxplot info`$names  
## [1] "Cat" "Dog" "Pig"

info[[1]]$`ANOVA summary info`

## Df Sum Sq Mean Sq F value Pr(>F)  
## Type 2 3.16 1.579 0.53 0.598  
## Residuals 17 50.64 2.979

etc...

# 4. Monte Carlo simulations

A Monte Carlo simulation is an algorithm that relies on repeated random sampling to obtain numerical results. In R, loops and functions are key for Monte Carlo simulations.

For example, we can simulate a die roll by taking a random sample from a 1:6 vector.

sample(1:6, 1)

## [1] 2

# We can also use the sample function to simulate 100 die rolls...  
sample(1:6, 100, replace=TRUE)

## [1] 2 5 3 5 3 5 6 3 5 6 2 4 1 2 3 1 3 6 3 3 4 3 2 5 5 5 1 5 3 5 4 5 4 4 5  
## [36] 1 3 5 5 3 6 3 2 1 1 2 4 4 3 6 2 3 2 4 2 3 5 1 6 3 6 3 3 3 6 6 3 5 6 3  
## [71] 5 3 2 5 2 5 1 2 1 2 1 4 6 5 5 3 3 5 4 4 3 2 6 4 2 1 3 6 4 6

But what if we wanted to repeat the process 200 times and get the mean of the die rolls for each iteration? One option is to do that with a loop:

iter <- 200  
nr\_rolls <- 100  
for (i in 1:iter){  
 rolls <- sample(1:6, nr\_rolls, replace=TRUE)  
 #print(rolls)  
 print(mean(rolls))  
}

Another alternative is to write a function that produces one iteration of the process and then use the replicate command to repeat the process 100 times.

## 4. Monte Carlo simulations - Step 1

Write a function that works through the process once

die\_roll\_mean <- function(nr\_rolls){  
 rolls <- sample(1:6, nr\_rolls, replace=TRUE)  
 mean(rolls)  
}  
  
die\_roll\_mean(nr\_rolls = 100)

## [1] 3.56

## 4. Monte Carlo simulations - Step 2

Now we use replicate to iterate the function 200 times:

?replicate

reps <- replicate(200, die\_roll\_mean(nr\_rolls = 100))

reps

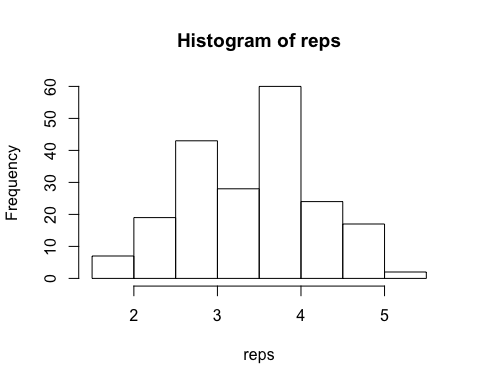
reps <- replicate(200, die\_roll\_mean(nr\_rolls = 5))

reps

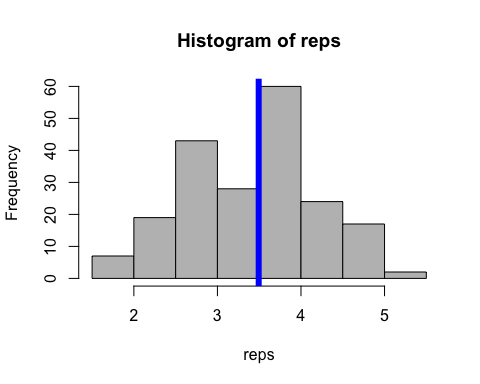
## 4. Monte Carlo simulations - include a plotting function

Again, we can use functions with plotting functions as well:

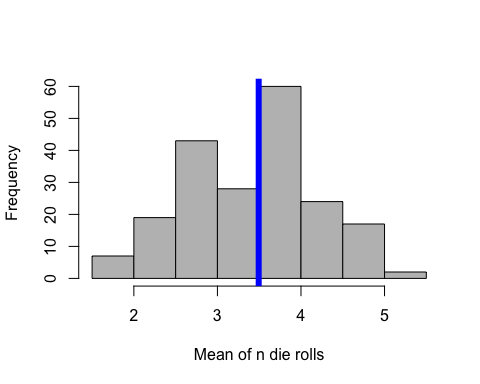
hist(reps)



hist(reps, col="grey")  
abline(v=mean(reps), col="blue", lwd=6)



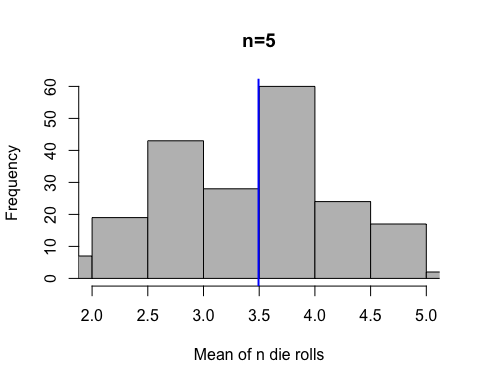
my\_hist <- function(sims){  
 hist(sims, col="grey", xlab="Mean of n die rolls", main="")  
 abline(v=mean(sims), col="blue", lwd=6)  
}  
  
my\_hist(sims=reps)



## 4. Monte Carlo simulations - Step 3

How would we update the function to add the number of die rolls as the title of the plot?

my\_hist <- function(sims, n){  
 hist(sims, col="grey", xlab="Mean of n die rolls",  
 main=paste0("n=",n), xlim=c(2,5))  
 abline(v=mean(sims), col="blue", lwd=2)  
 }  
  
my\_hist(sims=reps, n=5)

 How can we use everything we learned today to analyze the changes in the the sampling distribution of the mean of the die rolls as the number of die rolls change?

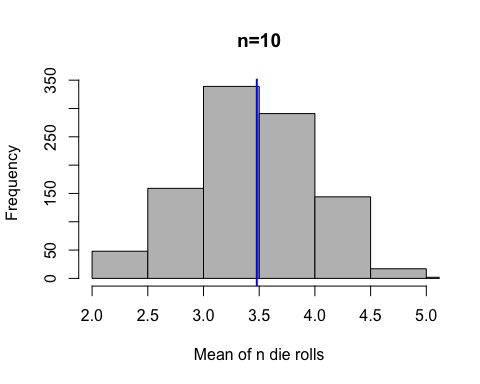
We can integrate what we did above and write a loop that varies the number of die rolls over which we take the mean and plots the sampling distribution each time.

Say we want to see the sampling distribution for , , , , , and .

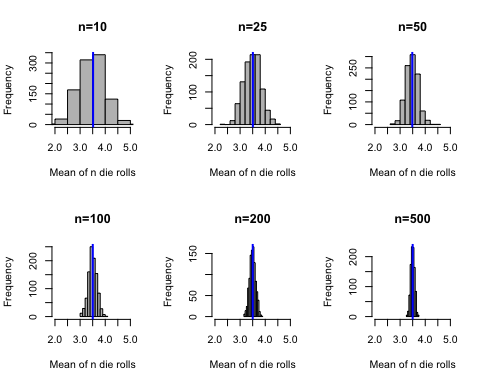
nr\_die\_rolls <- c(10, 25, 50, 100, 200, 500)

Let's see how we can first write this for . Remember this is the first element of the vector, so we can get it using 1 as the index, nr\_die\_rolls[1].

reps <- replicate(1000, die\_roll\_mean(nr\_rolls = nr\_die\_rolls[1]))   
my\_hist(sims=reps, n=nr\_die\_rolls[1])

 We can now write a loop that works through the vector with the numnber of die rolls and produces the relevant plot for each iteration:

par(mfrow=c(2,3))  
  
for (i in 1:length(nr\_die\_rolls)){  
 reps <- replicate(1000, die\_roll\_mean(nr\_rolls = nr\_die\_rolls[i]))  
 my\_hist(sims=reps, n=nr\_die\_rolls[i])  
}



# Challenge 4: The Birthday Problem

Say we choose 25 people at random. What is the probability two or more of them have the same birthday?'

Model simplifications: \* Ignore leap years \* Assume the probability of having a birthday in any of the 365 days of the year is equally likely (and no twins, triplets, etc. in the room).

Let's start by thinking of one room. We need to sample 25 birthdays and check how many are repeated.

set.seed(94705)   
birthdays <- sample(1:365, 25, replace=TRUE)  
birthdays

## [1] 135 225 191 242 351 276 10 214 330 288 242 322 129 234 21 86 77  
## [18] 17 103 46 23 289 85 54 174

We now want to know how many of those birthdays are repeated. For that we can use the unique() command.

unique(birthdays)

## [1] 135 225 191 242 351 276 10 214 330 288 322 129 234 21 86 77 17  
## [18] 103 46 23 289 85 54 174

length(unique(birthdays))

## [1] 24

23 birthdays are unique. Two people have the same birthday!

Note this is just one realization of the process. To use R to approximate the probability we would need to repeat this process many, many times.

For this we can start by writing a function that goes through the process one time and then use the replicate function as we did above.

We start by writing the function:

set.seed(94705)  
birthday\_function <- function(people=25){  
   
 # we populate the room  
 birthdays <- sample(1:365, people, replace=TRUE)  
  
 # get the unique number of bdays  
 unique\_bdays <- length(unique(birthdays))  
  
 # and return a 1 if at least one bday is repeated.  
 as.numeric(unique\_bdays!=people)  
}  
  
birthday\_function(people=25)

## [1] 1

Thus, we have one repeat birthday!

Now we can use replicate to repeat the process 1,000 times:

set.seed(94705)  
many\_sims <- replicate(1000, birthday\_function(people=25))

many\_sims

To approximate the probability of at least one matching birthday, we can just take the mean of this vector:

mean(many\_sims)

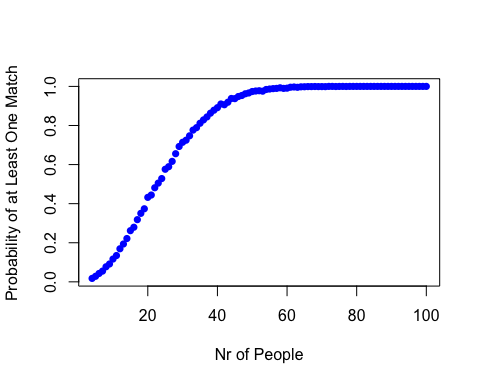
## [1] 0.578

Say we now want to use R to see how this probability changes as the number of people in the room changes. How can we do this? We can use our function but incorporating it into a loop which varies the number of people in the room:

people <- 4:100  
sims <- matrix(NA, length(people), 2)  
  
for(i in 1:length(people)){  
 many\_sims <- replicate(5000, birthday\_function(people=people[i]))  
 sims[i,] <- c(people[i], mean(many\_sims))  
}  
  
head(sims)

## [,1] [,2]  
## [1,] 4 0.0176  
## [2,] 5 0.0284  
## [3,] 6 0.0428  
## [4,] 7 0.0550  
## [5,] 8 0.0770  
## [6,] 9 0.0914

par(mfrow=c(1,1))  
plot(sims[,1], sims[,2], pch=16, col="blue",   
 xlab="Nr of People", ylab="Probability of at Least One Match")

 Learn more about the birthday problem [here](http://mathforum.org/dr.math/faq/faq.birthdayprob.html)

# Acknowledgements

[Software Carpentry](https://swcarpentry.github.io/)  
[Hadley Wickham](http://adv-r.had.co.nz/)  
[more Hadley Wickham](http://r-pkgs.had.co.nz/)