Lesson 8 (week 10) - Functions and binomials!

YOUR NAME

2023-11-14

Notes for lesson[1]:

- case study using the binomial distribution
- review loops - for and while[2] - before getting started
- Part 2: functions

Part 1: Case study: Hospital readmission rates of acute ischemic stroke in California

Tools we will use

- Chapters covered: This covers some concepts from chapters 3 and 7.
- Calculating binomial probabilities with `dbinom()`
- Finding p-value for a binomial problem using `pbinom()`
- Performing a binomial test using `binom.test`
- Calculating proportion confidence intervals with the `binom` package functions.

Calculating binomial probabilities

Example 1 : what is the probability of six successes in 27 trials, where p of success is 0.25.

```
dbinom(x = 6, size = 27, prob = 0.25)
```

Practice by finding these probabilities:

- probability of sampling 5 female snakes from a population where 0.70 are female.

[1] Developed by Daphne Hansell ('24) and Dr. Bitarello
[2] Previous two labs.

```r
dbinom(x = 5, size = 5, prob = 0.70)
```

```
## [1] 0.16807
```

- probability of rolling a 7 out of 10 rolls during a game:

```r
dbinom(x = 7, size = 10, prob = 0.50)
```

```
## [1] 0.1171875
```

Example 2 : finding the probability of 2 out of 3 children in a household being born in the same month.

Remember the binomial coefficient:

$$\binom{n}{X} = \frac{n!}{X!(n-X)!}$$

, where X is the number of successes and n is the number of trials.

and then remember the full binomial equation:

$$P[X] = \binom{n}{X} p^X (1-p)^{n-X}$$

[3]

```r
#manually
#p of two children in one given month and 1 child not in the same month

p<-1/12*1/12*11/12


#but we need to take into account that there are different configurations that could lead
↪  to this. We can use choose() to give us the binomial coefficient

binom_coef<-choose(n=3,k=2)

#now multiply p and binom_coef to find that total probability

print(p)
print(binom_coef)
print(p*binom_coef)

# now try the much simpler method using dbinom()

dbinom(x = 2, size = 3, prob = 1/12)
```

Now practice:

[3]Note: we write these equations using a typesetting language called Latex. It's great and awesome but you don't need to know it, but now you know what this is.

- find the probability that 6 out of the 12 students in our class were born on the same season of the year.

```
#use dbinom()

dbinom(x = 6, size = 12, prob = 1/4)
```

```
## [1] 0.04014945
```

Binomial test:

```
#null: p=0.061
#alt: p != 0.061
#define your alpha before hand: i will go with alpha=0.01 as my criterion for rejecting
↪  the null.Conf level is 1-alpha.

binom.test(x = 10, n = 25, p = 0.061,alternative = "two.sided", conf.level=0.99)
```

Example: is finding 10 successes out of 25 trials different from expected number of successes if p of success is 0.061?

```
##
##  Exact binomial test
##
## data:  10 and 25
## number of successes = 10, number of trials = 25, p-value = 9.94e-07
## alternative hypothesis: true probability of success is not equal to 0.061
## 99 percent confidence interval:
##  0.1678635 0.6702072
## sample estimates:
## probability of success
##                    0.4
```

Pratice:

- test if the die is biased: I rolled my personal super special die 80 times during a boardgame and got a 6 in 19 of those. My buddies accuse me and say my die is perhaps too good to be true. I plead innocence and boast about my cosmic luck! What do you think?

```
#null: ?
#alt: ?
#define your alpha:

binom.test(x = 19, n = 80, p = 1/6,alternative = "two.sided")
```

```
##
##  Exact binomial test
##
## data:  19 and 80
```

```
## number of successes = 19, number of trials = 80, p-value = 0.09805
## alternative hypothesis: true probability of success is not equal to 0.1666667
## 95 percent confidence interval:
##  0.1494538 0.3457770
## sample estimates:
## probability of success
##                 0.2375
```

Calculating confidence intervals of proportions

Example: What is the Agresti-Coull 95% confidence interval for a proportion based on 131 successes out of 169 trials?

`` `binom::binom.confint(131, n = 169, method = "ac")` ``

Motivation

According to the Nationwide Readmissions Database of the Healthcare Cost and Utilization Project between 2010 and 2015, the 30-day hospital readmission rate for acute ischemic stroke patients on a national level is 12.4% (Bambhroliya et al. 2018). A researcher wants to test whether the proportion of 30-day hospital readmissions for a California hospital with an "as expected" hospital quality rating differs from the national 30-day readmission proportion.

Data

CSV Data file: *readmin.csv*

The data file contains ischemic stroke 30-day hospital readmission incidence data for a random sample of patients in a California hospital with an "as expected" quality rating obtained from a set of hospital records for 2014-2015. The 30-day readmission data from a sample of 50 patients was recorded.

Read in the dataset (it's in the input_files folder) which we'll call *readmin*. Use `head` to check what it looks like:

```
# read dataset
# use head()
```

The variable *ReadmissionStatus* is a binary variable which is equal to 1 if the patient was readmitted to the hospital within 30 days of discharge and equal to 0 if the patient was not readmitted to the hospital within 30 days of discharge.

Questions of interest

Two questions of interest are:

1. Is the proportion of stroke patients readmitted within 30-days of discharge from CA hospital different from the nationwide proportion?
2. What is the 95% confidence interval for the proportion of acute ischemic stroke patients readmitted within 30 days of discharge?

Instructions

Ex2.1: Install the `binom()` package and load it:

```
# Install
#install.packages("binom")
# load package
library(binom)
```

Exploring the Data

Let's begin by looking at the variable of interest, readmission. Use the `table()` function.

```
# Table of Readmission Status
```

How many peope are readmitted to the hospital within 30 days? How many are not?

Data Analysis

Determine the proportion of individuals readmitted in the sample. *Tip: check out the* ***`prop.table()`*** *function.*

```
# Proportion table of Readmission Status
```

Answer: of the sample was readmitted within 30 days.

The Binomial test Write the null and alternative hypotheses for this statistical test, comparing the proportion of readmission in our study, to the population proportion of 0.124.

H_0 : The proportion of readmission in the population is _____

H_A : The proportion of readmission in the population is not _____

Run a two-sided binomial test to see if the sample proportion differs significantly from the population proportion. *Tip: Some possibilities were covered in lectures. You can also just check out the* ***`binom`*** *package now.*

```
# Binomial test of readmission status
```

What is your decision based on the binomial test?

Answer:

Agresti-Coull 95% Confidence Interval for the proportion Next, calculate the 95% CI for the proportion of stroke patients who were readmitted within 30 days of discharge using the Agresti-Coull method. *Tip: Some possibilities were covered in lectures. You can also just check out the* ***`binom`*** *package now.*

```
# 95% CI for binomial test
```

What is the Agresti-Coull 95% CI? Answer:

Answer:

Conclusions

Based on these results, do we find evidence that the population proportion is significantly different from 0.124? Use both the test and the CI to answer this.

Answer: we find no evidence that the population proportion is significantly different from 0.124. The 95% CI interval ($0.0525 < p < 0.2417$) indicates medium level of precision, given that it covers nearly 20% of possible proportions.

References

Bambhroliya AB, Donnelly JP, Thomas EJ, et al., "Estimates and Temporal Trend for US Nationwide 30-Day Hospital Readmission Among Patients With Ischemic and Hemorrhagic Stroke.", *JAMA Netw Open*, 1[2018]:e181190 https://jamanetwork.com/journals/jamanetworkopen/fullarticle/2696869

Functions

Today we are going to learn about functions! I am going to quote directly from the R god Hadley Wickham:

"Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting. Writing a function has four big advantages over using copy-and-paste:

1. You can give a function an evocative name that makes your code easier to understand.
2. As requirements change, you only need to update code in one place, instead of many.
3. You eliminate the chance of making incidental mistakes when you copy and paste (i.e. updating a variable name in one place, but not in another).
4. It makes it easier to reuse work from project-to-project, increasing your productivity over time."

If this tutorial makes no sense to you, you can go read his.[4]

All the fun stuff you do with the `tidyverse` packages are just functions **someone else** built and published as a package. And now you are going to learn to do exactly the same thing.

So what is in a function?

Components:

- Function Name: How we will refer to our function.
- Arguments: Variables passed into the function.
- Function Body: The code that performs the task.
- Return Statement: The output of the function.

[4] https://r4ds.hadley.nz/functions

```
# skeleton of a function
functionName <- #function name
        function(arg1, arg2, ...) { #arguments
         # Function Body
         # ...
        return(result) #return statement
}
```

Example: The function `addNumbers` adds 2 numbers together and returns the output.

```
addNumbers <- function(num1, num2) { #function name and arguments
  sum <- num1 + num2 #function body
  return(sum) #return statement
} #close the curly brackets!
```

Run it in your console and then try this code:

```
addNumbers(12, 865)
```

```
## [1] 877
```

Woah look at that! It added the numbers and printed the result. You could try again with other numbers but I think you get the idea.

Ex.1: Can you now make a basic function that multiplies two numbers and returns the result? Call it something like `multiplyNumbers`.

```
# function name and arguments
mulitplyNumbers<- function(num1, num2) {
  #function body
  product <- num1 * num2
  #return statement
  return(product)
}
```

Try it with some numbers! You wrote your very first function!!!

Ok, these seem fairly useless – obviously R can add and multiply pretty easily without this. But lets say you're a budding young biologist who has to work with Celsius in their lab but wants to know what the temperatures are in Fahrenheit. You are sick of constantly googling the conversion, so you decide to automate it in R.

Here's the equation to convert Celsius (centigrade) to Fahrenheit.

$$^0F = (1.8 \times ^0 C) + 32$$

Ex.2: Write a simple function that takes in a value in Celsius and converts it into Fahrenheit.

```
#call your function CtoF or anything meaningful to you
#give it one argument called "c"


#In the body of the function, write out the math of the conversion.
#Use the formula above to create a variable "f" to store the converted temperature.


# in the return statement, return "f", your newly created variable. Note, this variable
↪  only exists inside your function.


CtoF<-function(c){
f<- (c *1.8) + 32
return(f)
}
```

Ex.3: Now you can change your temperatures and convert them easily! Test your function to do the following conversions from Celsius to Fahrenheit:

- 0 degrees Celsius (water freezing point at sea level)
- 100 degrees Celsius (water boiling point at sea level)
- − 273.15 degrees Celsius (absolute zero: the lowest limit of the thermodynamic temperature)

```
# 0 degrees Celsius (water freezing point at sea level)
CtoF(c=0)
```

```
## [1] 32
```

```
# 100 degrees Celsius (water boiling point at sea level)
CtoF(c=100)
```

```
## [1] 212
```

```
# - 273 degrees Celsius (absolute zero: the lowest limit of the thermodynamic
↪  temperature)
CtoF(c=-273.15)
```

```
## [1] -459.67
```

Ex.4: Here's the equation to convert Fahrenheit to Celsius.

$$^0C = \frac{^0F - 32}{1.8}$$

Let's try the same thing but the other way around.

Write a simple function that takes in a value in Celsius and converts it into Fahrenheit.

```r
#call your function FtoC or anything meaningful to you
#give it one argument called "f"


#In the body of the function, write out the math of the conversion.
#Use the formula above to create a variable "c" to store the converted temperature.


# in the return statement, return "c", your newly created variable. Note, this variable
↪  only exists inside your function.

FtoC<-function(f){
c<- (f-32)/1.8
return(c)
}


# now test that running FtoC() on the temperatures you obtained with CtoF gives the
↪  expected outputs:

FtoC(f=32)
```

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

```r
FtoC(f=212)
```

```
## [1] 100
```

```r
FtoC(f=-459.67)
```

```
## [1] -273.15
```

What if we wanted to combine these functions to make one big temperature conversion function that could go either way (Celsius-> Fahrenheit or Fahrenheit->Celsius)? For that, you'd have to know how to use conditional statements in a function. And you do! (sort of)

Conditional Statements in Functions

It is useful to have functions that can do different things depending on the input.

For example, if we had a big temperature conversion function you would want it to differentiate Fahrenheit from Celsius and do different math depending on which you gave it.

For example the function below tells you if a number is even. If the number is even, it returns `TRUE`. If it is odd, it returns `FALSE`.

```r
isEven <- function(number) { #function name and arguments
  if (number %% 2 == 0) { #body of the function. If remainder is zero when nr/2
    return(TRUE) # then it is even
  } else { #if remainder when nr/2 is NOT 0
```

```
    return(FALSE) #nr is odd.
  }
}
```

Run it and try some numbers and see that it does what you think it does. For the following numbers, first think of what output you expect and then run the function with that value to confirm your prediction:

- 242
- -3
- 75
- -20

Now make the reverse of this function called `isOdd`. If the number is not divisble by 2 it should return `TRUE`, and if the number is divisible by 2 it should return `FALSE`

```
isOdd <- function(number) { #function name and arguments
  if (number %% 2 == 0) { #body of the function. If remainder is zero when nr/2
    return(FALSE) # return FALSE
  } else { #if remainder when nr/2 is NOT 0
    return(TRUE) #return TRUE
  }
}
```

Run this one also on the same examples as above, but first predict what the function should return in each case.

Conditional statements recap

You can see how useful conditional statements are once you can automate them with functions! Remember how you would set your variable to a value, run the conditional then go back and change the original variable if you wanted to do it again? Like this example from lesson 7:

```
number <-3

if (number > 0) {
  print("The number is positive.")
} else if (number < 0) {
  print("The number is negative.")
} else {
  print("The number is zero.")
}
```

```
## [1] "The number is positive."
```

Try using the logic of the function above to write a function called `classifyNumber` that uses an input number and returns a statatement (printed message) telling you whether: it's positive, negative or 0. Hint: You can copy and paste that code and just edit it.

```
#call your function classifyNumber and give it one argument called "number"
classifyNumber<-function(number){ #function name and arguments
if (number > 0) { #if number is >0
```

```
  print("The number is positive.") #print a statement saying that.
} else if (number < 0) { #otherwise, if number <0
  print("The number is negative.") # print a statement saying that.
} else { #if neither of the above is true
  print("The number is zero.") #print a statement saying what it is
}
}

#make sure you test your function (always).
#test it with these numbers, for example: 2, 213213, -100, 0.00000001
```

Pass vectors as argument values

Ex.6: So far we've been providing single numbers to our aguments, but any function that works for one number should also work for a vector of numbers.

For example, try out your functions `FtoC`, `CtoF`, `classifyNumber` using vectors with multiple values. Note what the output looks like. Does it make sense to you?

- `FtoC` on vector `c(0,100, -273.15)`
- `CtoF` on vector `c(32, 212, 459.67)`
- `classifyNumber` on vector `c(2, 213213, -100, 0.00000001)`

```
#`FtoC` on vector `c(0,100, -273.15)`


#`CtoF` on vector `c(32, 212, 459.67)`


#`classifyNumber` on vector `c(2, 213213, -100, 0.00000001)`
```

Mulitple Arguments

So far all of our functions have taken just one or two arguments.

Ex.7: Try running them without an input. Try this with `FtoC`, `CtoF`, and `classifyNumber` in two ways: a) without passing any arguments to the function; b) passing a value of `NULL` to the argument required by the function:

```
#`FtoC


#CtoF


#classifyNumber
```

- In the first case, you get an error that the argument is missing and has no default.
- In the second case, you get an error saying the argument is of length zero (because you gave the argument the `NULL` value).

Many of the functions you use regularly have a bunch of arguments that you could customize but don't have to. Why? They have default values that R uses unless you tell it to do something else. For example, type ?mean into your console and read the "arguments" portion of the help page. You probably only ever include the first argument, the values you want the mean of. But you have the option to exclude some percentage of the values or remove missing data (which we have done before). The values listed just above "arguments" in the help page indicate what the assumed default values are. For `mean`, these are: `x`, `trim=0`, `na.rm=FALSE`. If you run `mean(c(0,NA,3))` the function returns `NA` because `na.rm=FALSE` by default. But you can change this by setting it to true: `mean(c(0,NA,3), na.rm=TRUE)`, which returns the mean of 0 and 3, i.e., 1.5.

Below is a function that raises a number to the nth power.

```
powerraiser<-function(x, power){ #x and power are the arguments
        result<-x^power #body of the function
        return(result) #what the function returns
}
```

If we try to run this function without specifying the arguments, we will get errors:

```
#try running these

#neither argument is defined with no default to revert to
powerraiser()


#only one argument is defined with no default to revert to
powerraiser(x=2)

#only one argument is defined with no default to revert to
powerraiser(power=3)
```

Ex.8: If we wanted to write a function that raises numbers to the nth power but we think we will mostly use it for squaring numbers, we could write it with a default value for the power argument. Do that below:

```
#call your function powerraiser_v2 to differentiate it from the previous powerraiser
↪  function
powerraiser_v2<-function(x, power = 2){
        result<-x^power
        return(result)
}

# try running it by passing a value to x but not to power

powerraiser_v2(x=2)
```

```
## [1] 4
```

```
# try running it by passing a value to x and a value that is not 2 to power

powerraiser_v2(x=2, power = 3)
```

```
## [1] 8
```

You can also have the default for an argument be `null`. If you set the default to null, your function will simply ignore that argument.

Example: Lets say you want to make a function that calculates the area of a rectangle. Your arguments would be `length` and `width`.

```
dummy_function<-function(length, width){
        area = length * width
        return(area)
}

# test it. All the runs below mean the same thing:

dummy_function(length = 2, width = 3)
```

```
## [1] 6
```

```
dummy_function(2, width = 3)
```

```
## [1] 6
```

```
dummy_function(length = 2, 3)
```

```
## [1] 6
```

```
dummy_function(2, 3)
```

```
## [1] 6
```

Be careful, though, when you don't spell out the arguments as we did above: `dumny_function(2,3)` instead of `dummy_function(x = 2, power = 3)`. In those cases, R assumes you are passing arguments in the order in which they are defined in the function.

Below is an updated function of `dummy_function` which prints out what the `length` and `width` parameter values are, as well as the calculated `area`:

```
dummy_function_v2<-function(length, width){
        mess1<-paste("Length is:", length)
        mess2<-paste("Length is:", length)
        print(mess1)
        print(mess2)
        area = length * width
        mess3<-paste("Area is:", area)
        return(area)
}
```

Ex.9: Think about what you think will happen if you run `dummy_function_v2` in the following ways, always with the goal of assigning the value 4 to width and the value 2 to length:

- a. `(width = 4, length = 2)`

- b. `(length = 2, width = 4)`
- c. `(4,2)`
- d. `(2,4)`

What will be the area in each case? What will the different print messages print to your console? Now check that the results meet your expectations:

```
#a

#b

#c

#d
```

What did you observe? Are the areas always the same? Are the parameter values passed to the right parameter variables?

Answer: because the area of a rectangle is the same whether we define `width = 2, length = 4` or `width = 4, length = 2`, running a-d gives us the same area

Conclusion: when you name arguments explicitly you have the flexibility of not needing to put them in a specific order inside the function command. Also, you make things more explicit to yourself and whoever is reading your code.

Let's build on top of this. If you wanted to calculate the area of a square using this same function, you would only need to enter one value, since for a square both dimensions are the same. Here is the list of steps we will need to write this function.

- Write a function that takes in both arguments (length and width) – call it `calculate_area`
- The default values for both arguments should be NULL – `(width = NULL, length = NULL)`

```
calculate_area<-function(width = NULL, length = NULL){
        #body of the function


}


#try to run this function without specifying argument values.
#you get something not very helpful: NULL. That's because you didn't actually do anything
↪  in the function yet.

calculate_area()
```

```
## NULL
```

- The body of the function checks whether both of them are null. That would be the case if the user did not specify either parameter when using the function – a silly think to do!
- we will use the `is.null()` function to check if both parameters are NULL. We will use & to combine both statements since we are checking if both are NULL, not just one of them

- If that is the case (that both are null), we want to tell the user they did something very wrong and that the function should just stop right there. We can do this using the `stop` command.
- Have a look at `?stop`. Look at the examples at the bottom.

```
calculate_area<-function(width = NULL, length = NULL){
        #body of the function
        #check if both parameters are null. If so, halt and print the error message.
        if(is.null(width) & is.null(length)){
                stop("Both width and length are undefined. Seriously, what do you hope
                ↪  will happen here sweety?")
        }

}

#test that your function does what you think it does.
#eg.

calculate_area()
```

Great! Moving on.

- Otherwise (if it is NOT true that both parameters are null), we could ask if one of them is, but not both
- the first `if` statement already checks if both of them are null. If that is true, the function prints the error message.
- now we can add an `else if` statement that asks whether one of the two is null but not both.
- if that is true, we want the area to be calculated as the square of the non-null parameter.

```
calculate_area<-function(width = NULL, length = NULL){
        #body of the function
        #check if both parameters are null. If so, halt and print the error message.
        if(is.null(width) & is.null(length)){
                stop("Both width and length are undefined. Seriously, what do you hope
                ↪  will happen here sweety?")
        }else if (is.null(width)){
               area <- length ^ 2
        }

        return(area)
}

#test that your function does what you think it does.
#eg.should give you 2^2=4 as an output

calculate_area(length = 2)

#should give an error sayign object area cannot be found. It can't! Because your script
↪  did not create the object area in this case.
calculate_area(width = 2)
```

- Now let's add another code chunk for the case where width is not null, but length is. Since either dimension is sufficient for the area of a square, we can use that as well in the same way:

```
calculate_area<-function(width = NULL, length = NULL){
        #body of the function
        #check if both parameters are null. If so, halt and print the error message.
        if(is.null(width) & is.null(length)){
                stop("Both width and length are undefined. Seriously, what do you hope
                ↪  will happen here sweety?")
        }else if (is.null(width)){
               area <- length ^ 2
        }else if (is.null(length)){
                area <- width ^ 2
        }

        return(area)
}

#test that your function does what you think it does.
#eg.this should give you 2^2=4 as an output

calculate_area(length = 2)
```

```
## [1] 4
```

```
#eg. this should give you 2^2=4 as an output
calculate_area(width = 2)
```

```
## [1] 4
```

- Let's add a final chunk that produces the `area` in the case where both width and length are defined.
- We will add this as a final `else` statement.

```
calculate_area<-function(width = NULL, length = NULL){
        #body of the function
        #check if both parameters are null. If so, halt and print the error message.
        if(is.null(width) & is.null(length)){
                stop("Both width and length are undefined. Seriously, what do you hope
                ↪  will happen here sweety?")
        }else if (is.null(width)){
               area <- length ^ 2
        }else if (is.null(length)){
                area <- width ^ 2
        }else{ #if all the previous statements are false, the only remaining possibility
↪  is that neither is null.
                area <- width * length

        }

        return(area)
}

#test that your function does what you think it does.
#eg.this should give you 2*4=8 as an output
```

```
calculate_area(length = 2, width = 4)
```

```
## [1] 8
```

- lastly, we can add some nice messages telling the user what is going on:

```
calculate_area_final<-function(width = NULL, length = NULL){
        #body of the function
        #check if both parameters are null. If so, halt and print the error message.
        if(is.null(width) & is.null(length)){
                stop("Both width and length are undefined. Seriously, what do you hope
                ↪  will happen here sweety?")
        }else if (is.null(width)){
               area <- length ^ 2
               mess<-paste("Width wasn't defined. The area of the square is: ", area)
               print(mess)
        }else if (is.null(length)){
                area <- width ^ 2
                mess<-paste("Width wasn't defined. The area of the square is: ", area)
                 print(mess)
        }else{ #if all the previous statements are false, only remaining possibility is
↪  that neither is null.
                area <- width * length
                mess<-paste("Both width and length were defined. The area of the
↪  rectangle is:", area)

        }

        return(area)
}

#test that your function does what you think it does.

test1<-calculate_area_final(width = 2)
```

```
## [1] "Width wasn't defined. The area of the square is:  4"
```

```
test2<-calculate_area_final(length = 2)
```

```
## [1] "Width wasn't defined. The area of the square is:  4"
```

```
test3<-calculate_area_final(length = 2, width = 4)

print(test1)
```

```
## [1] 4
```

```r
print(test2)
```

```
## [1] 4
```

```r
print(test3)
```

```
## [1] 8
```

```r
#first name your function "calculate_area" and add two arguments: length and widtg.
# set width equal to NULL
calculate_area <- function(length, width = NULL) {
#then add a conditional statement: If width is NUll, set width equal to length.
  if (is.null(width)) {
    width <- length
  }
 #then calculate the area and don't forget to return your result
 area <- length * width
  return(area)
}
```

Combining everything!

Now lets go back to our temperature functions from earlier.

```r
CtoF<-function(c){
f<- (c *1.8) + 32
return(f)
}

FtoC<-function(f){
c<- (f-32)/1.8
return(c)
}
```

We want to make our **ONE BIG FUNCTION**:

- call it `temp_conv` and define two parameters: `c` (celsius) and `f` (fahrenheit)
- set both default values for the parameters to be `NULL`
- use conditional statements to define what happens next: if provided temperature is in Celsius, convert it to fahrenheit and return the temperature in fahrenheit; if provided temperature is in Fahrenheit, convert it to Celsius and return the temperature in Celsius:

```r
# name function temp_conv
# define two arguments: c and f
# set both to default values of NULL
temp_conv<-function(c = NULL, f = NULL){
#conditional statement: if f is null that means you are converting from Celsius, so you
↪  should be returning Fahrenheit
if(is.null(f)){
```

```
        tf<- (c *1.8) + 32
        return(tf)
#otherwise, you are starting from Fahrenheit, so you should be returning Celsius
}else{
      c<- (f-32)/1.8
        return(c)
}
}
```

Test your function here. Before you run each command, try to anticipate what the output should look like. Get in the habit of checking that your function does what you want it to do:

```
temp_conv(c = 100) #test 1: should return 212 Fahrenheit
```

```
## [1] 212
```

```
temp_conv(c = 0) # test2L should return 32 Fahrenheit
```

```
## [1] 32
```

```
#try more values here
```
