Introduction to Programming with R

Control Flow and Functions

Selene R. Schmittling

2025-08-01

Learning Outcomes

In this Session of our R Workshop, we will cover the following introductory aspects of R Studio & Programming:

- Conditional and Flow Control Statements
  - if
  - for
  - while
- Functions, including
  - built-in
  - library
  - user-defined
- Getting help
- File Input and Output

By the end of this Session, you will understand:

1. How to write an if statement and what to use it for
2. How to write a for loop, how it works and what to use if for
3. How to write a while loop, how it works and what to use it for
4. The three types of functions
5. The anatomy of a function
6. How to install and use R Libraries
7. How to use documentation, get help in RStudio
8. How to import and save data

Conditional Statements and Flow Control

If Statement (Conditional)

The `if` statement allows us to run code based on some condition (**if** the condition is true, the code is run). The `if` statement can use a logical variable or an expression that evaluates to a logical value as its "test". Here is what it looks like:

```
eatsHamburger = TRUE
eatsVegees = TRUE
eatsMeat = TRUE
numHamburgersEaten = 1 # number per week
my.calorie.count <- 0


# Use relational operator
if (numHamburgersEaten <= 1) {
```

```
  eatsHamburger = TRUE
}

# Use a logical operator
if (eatsVegees==TRUE && eatsMeat==FALSE) {
  # I'm a vegetarian
  eatsHamburger = FALSE
}

# Use logical variable
if (eatsHamburger) {
  my.calorie.count <- my.calorie.count + 650
}
```

You use if statements when you want to execute code **conditionally**, e.g. only if certain conditions are met.

We can expand the `if` statement with `else`. The `else` clause is used to run code if the condition **isn't** met. Let's look at some examples:

```
eatsHamburger = TRUE
eatsVegees = TRUE
eatsMeat = TRUE
numHamburgersEaten = 1 # number per week
my.calorie.count <- 0

# increase calorie count
if (eatsHamburger) {
  my.calorie.count <- my.calorie.count + 650
} else {
  my.calorie.count <- my.calorie.count + 60 # I should choose this more often
}

# make sure you have open ({) and closed (}) braces! Let's remove one and see what R does. Also note, i
```

While we have provided examples with one line of code, you can put as much code between the two braces as you need.

For Loop

A `for` loop is used to perform code multiple times. The code in a `for` loop will, by definition, run a specified number of times. This is because you provide a range, in some form, over which the loop executes. This makes `for` loops very useful for traversing data structures like vectors, matrices, and dataframes. Let's look at some examples:

```
# Simplest example
for (i in 1:5) {
  print(i) # spice it up and divide i by 2
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
vector1 = c("one", "two", "three", "four", "five")
for (x in vector1) {
```

```
  print(x)
}
```

```
## [1] "one"
## [1] "two"
## [1] "three"
## [1] "four"
## [1] "five"
```

```
vector2 = c(2, 4, 6, 8, 10)
for (y in vector2) {
  print(y)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

In the last couple of examples, we looped over a vector. What about a matrix? In this case we would do nested `for` loops.

```
# Make a matrix
data = (1:60)
my.matrix = matrix(data, nrow=10, ncol=6, byrow = TRUE)
print(my.matrix)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
##  [1,]    1    2    3    4    5    6
##  [2,]    7    8    9   10   11   12
##  [3,]   13   14   15   16   17   18
##  [4,]   19   20   21   22   23   24
##  [5,]   25   26   27   28   29   30
##  [6,]   31   32   33   34   35   36
##  [7,]   37   38   39   40   41   42
##  [8,]   43   44   45   46   47   48
##  [9,]   49   50   51   52   53   54
## [10,]   55   56   57   58   59   60
```

```
# Lets print every value in the matrix out one at a time
# REMEMBER TO MAKE SURE YOU HAVE OPEN & CLOSED BRACES!
for (row in 1:10) {
  for (col in 1:6) {
    print(my.matrix[row,col])
  }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

```
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
## [1] 43
## [1] 44
## [1] 45
## [1] 46
## [1] 47
## [1] 48
## [1] 49
## [1] 50
## [1] 51
## [1] 52
## [1] 53
## [1] 54
## [1] 55
## [1] 56
## [1] 57
## [1] 58
## [1] 59
## [1] 60
```

```
# We can use functions to extract number of rows and cols
for (row in 1:nrow(my.matrix)) {
```

```
  for (col in 1:ncol(my.matrix)) {
    print(my.matrix[row,col])
  }
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
## [1] 35
## [1] 36
## [1] 37
## [1] 38
## [1] 39
## [1] 40
## [1] 41
## [1] 42
## [1] 43
## [1] 44
## [1] 45
## [1] 46
## [1] 47
## [1] 48
## [1] 49
```

```
## [1] 50
## [1] 51
## [1] 52
## [1] 53
## [1] 54
## [1] 55
## [1] 56
## [1] 57
## [1] 58
## [1] 59
## [1] 60
```

```
# How would you use this same kind of structure to access data in a 3-D array?
```

While Loops

While loops execute code by testing a condition. This means that the code may execute once, several times or not at all! While loops are great when you need to perform some tasks, but you don't know how many times. Humans do this all the time, "I want to pet my dog until the dog hair on my pants makes them look like fur leggings." You don't know when that condition will happen, but when it does you want to stop petting your dog.

While `for` loops execute over a range of values, `while` loops need a condition to test. This means you use a logical variable, relational expression or logical expression to determine if the code should run. Here are some examples:

```
# monitor our loop using this logical variable
eatsHamburger = TRUE

# Broken While loop
# Uncomment and run, but comment it back before rendering notebook!
# numBurgersEaten <- 0
# my.calorie.count <- 0
# eatsHamburger <- TRUE
# while (eatsHamburger == TRUE) {
#   my.calorie.count <- my.calorie.count + 650
#   numBurgersEaten <- numBurgersEaten + 1
#   print(numBurgersEaten)
# }


#Ok, let's try this again
numBurgersEaten <- 0
my.calorie.count <- 0
eatsHamburger <- TRUE
while (eatsHamburger == TRUE) {
  my.calorie.count <- my.calorie.count + 650
  numBurgersEaten <- numBurgersEaten + 1
  if (numBurgersEaten > 5) {
    eatsHamburger <-  FALSE # OK we've added code to change value of condition
  }
  print(numBurgersEaten)
  print(my.calorie.count)
  print(eatsHamburger)
}
```

```
## [1] 1
## [1] 650
## [1] TRUE
## [1] 2
## [1] 1300
## [1] TRUE
## [1] 3
## [1] 1950
## [1] TRUE
## [1] 4
## [1] 2600
## [1] TRUE
## [1] 5
## [1] 3250
## [1] TRUE
## [1] 6
## [1] 3900
## [1] FALSE
```

You are just starting out, but let's talk about efficiency when you are writing code. There will be many ways to write code. Good code is

- code that works and
- that can be understood by others.

Once you are good at that, and as your experience grows, you will work on efficient. Efficiency comes with practice and in exploring different ways to write the same code.

Let's look at how we might make the code above more efficient.

```
# Ok, let's look at the previous code
numBurgersEaten <- 0
while (eatsHamburger == TRUE) {
  my.calorie.count <- my.calorie.count + 650
  numBurgersEaten <- numBurgersEaten + 1
  if (numBurgersEaten > 5) {
    eatsHamburger == FALSE # OK we've added code to change value of condition
  }
}

# Walk through the steps in your head. Is there anything that seems duplicative? Focus on the "if" stat

# What if we just use numBurgersEaten as our "while" test?
# How you you rewrite it to use that instead?

numBurgersEaten <- 0
while (numBurgersEaten <= 5) {
  my.calorie.count <- my.calorie.count + 650
  numBurgersEaten <- numBurgersEaten + 1
}
```

Some important considerations with `while` loops is that you need to ensure:

- you initialize any values used inside the loop to change the condition BEFORE the while loop starts
- that the condition used to determine whether the while loop is run again is updated within the while loop

I digress ...

Sometimes you need to exit an `if`, `for`, or `while` loop before all the code is executed. Most languages, including R, have a reserved word you can use to exit a loop, if needed. This word is `break`. While we won't cover it today, it is an important enough concept that I want to provide an example.

```
# Use a for loop as an example, but this will work with if and while statements too

for (i in 1:10) {
  print(paste("Current value:", i))

  # if value = 7 print a different message and exit loop
  if (i == 7) {
    print("Elvis has left the building. Bye!")
    break # this statement will end the loop change 7 to another number
  }
}
```

```
## [1] "Current value: 1"
## [1] "Current value: 2"
## [1] "Current value: 3"
## [1] "Current value: 4"
## [1] "Current value: 5"
## [1] "Current value: 6"
## [1] "Current value: 7"
## [1] "Elvis has left the building. Bye!"
```

Functions

We will look at three types of functions:

- Built-in
  - are available when R loads without further info
- External package functions
  - need to be initially installed
  - when you want to use them you must load them using the `library()` command
- User-defined functions

We'll talk about each of these and explore some examples. The following vocabulary will be helpful:

> **define**: before they can be used, functions need to be defined. For built-in and external package functions this will already be done for you (sweet!!). However, as the name suggests when you create your own functions (user-defined) you will "define" the function
>
> **call**: when you use a function in your program you "call" it.

A function has three main components:

- name (so you can refer to it because "hey, you!" won't get its attention)
- parameters
  - pieces of information the function needs to do it's job.
  - parameters can be required or optional
    - the function won't run without required parameters

        * optional parameters will have a default value which is used if no value is passed for it
        * the term parameters refers to the variables the function needs to do its job.
        * when we provide values for those parameters, we call them arguments.
- return value
    - this is a value that the function returns when it's done
    - frequently we will store this value in a variable using the assignment operator
    - the return value may be
        * a value that was calculated
        * data that has been manipulated in some way, or
        * a status that indicates whether the function encountered any errors

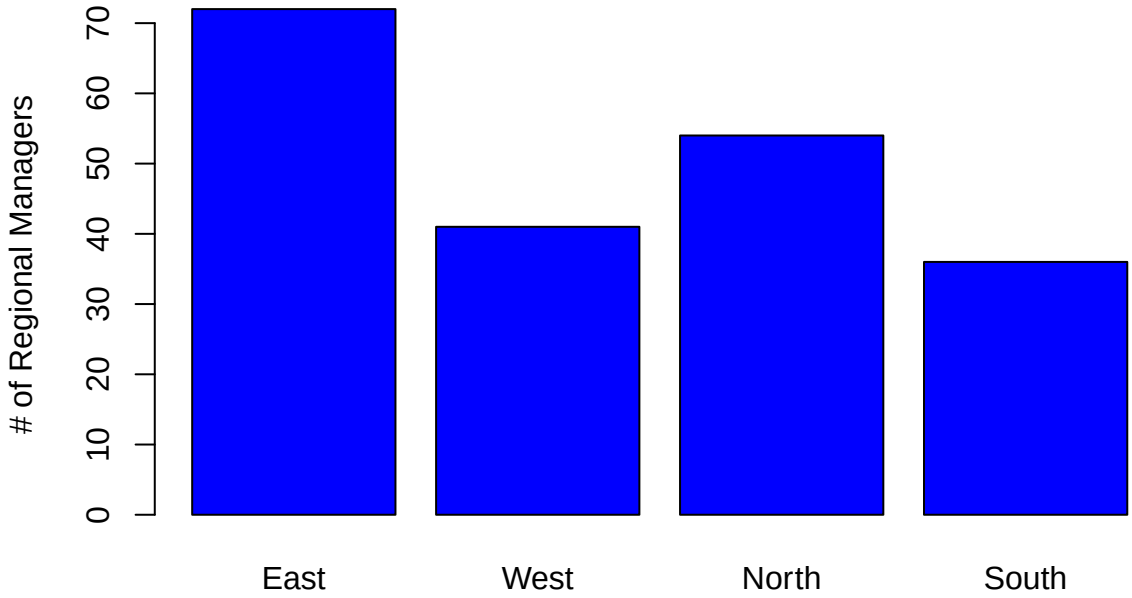
Built-In

Built-in functions are functions that are loaded when R is loaded. You don't have to do anything to use them.

You can see what packages R loads by opening the Packages tab in the Output pane. If you haven't loaded any other packages, then the packages with check-marks are the built-in packages that get loaded.

We saw an example of using built-in package in the Basic Syntax and Operators Workshop.

```
# define some data
data <- data.frame(a = c(72,41,54,36), b=c('East','West','North','South'))

# generate plot
barplot(data$a,
        names.arg = data$b,
        col="blue",
        ylab="# of Regional Managers")
```



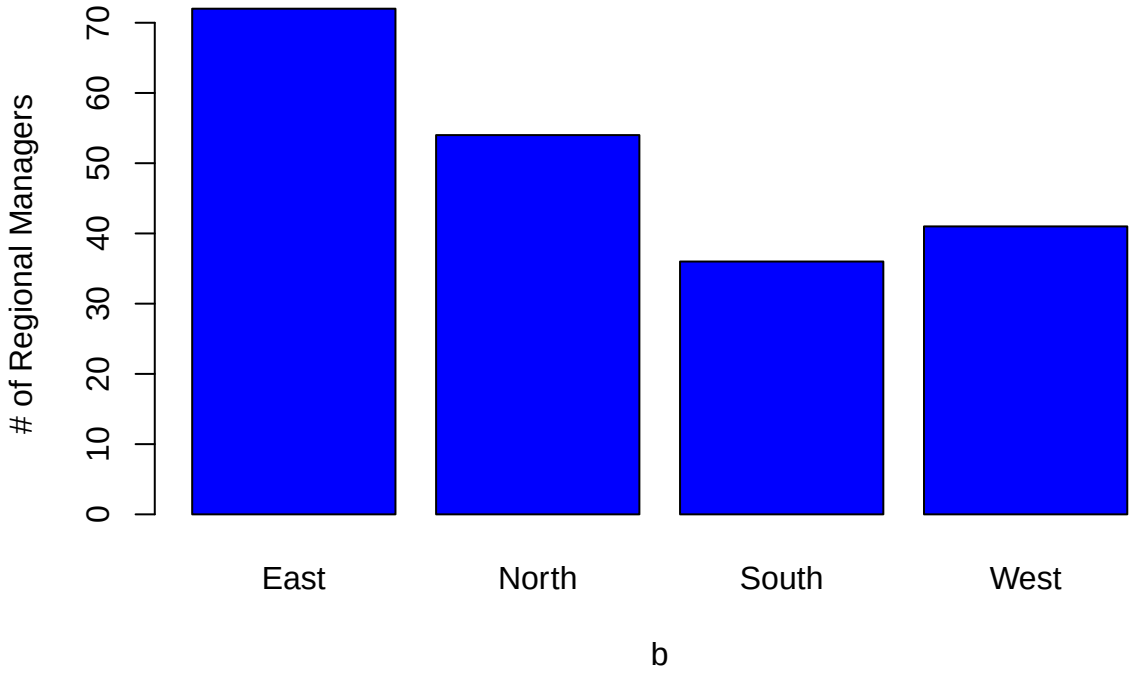
The `barplot()` function is part of Base R and is loaded by default. In the console window, type "?barplot" and hit enter. Let's use the vocabulary we covered earlier and read some documentation!

Here is what we see in the Help pane:

- name of the function and the package it is in: barplot {package}
- title of the help page: Bar Plots

- Description: gives a high-level description of what the function does
- Usage:
  - provides the function interface
    - all parameters, required and optional
      - optional parameters will be followed by an "=" sign
    - different "flavors" of the function
- Arguments
  - lists all parameters
  - sometimes indicates "optional", but not always
  - includes acceptable values, data types and data structures
- Value
  - provides information about the return value
- Authors, References, See Also and Examples
  - self-explanatory
  - click on "Run examples"

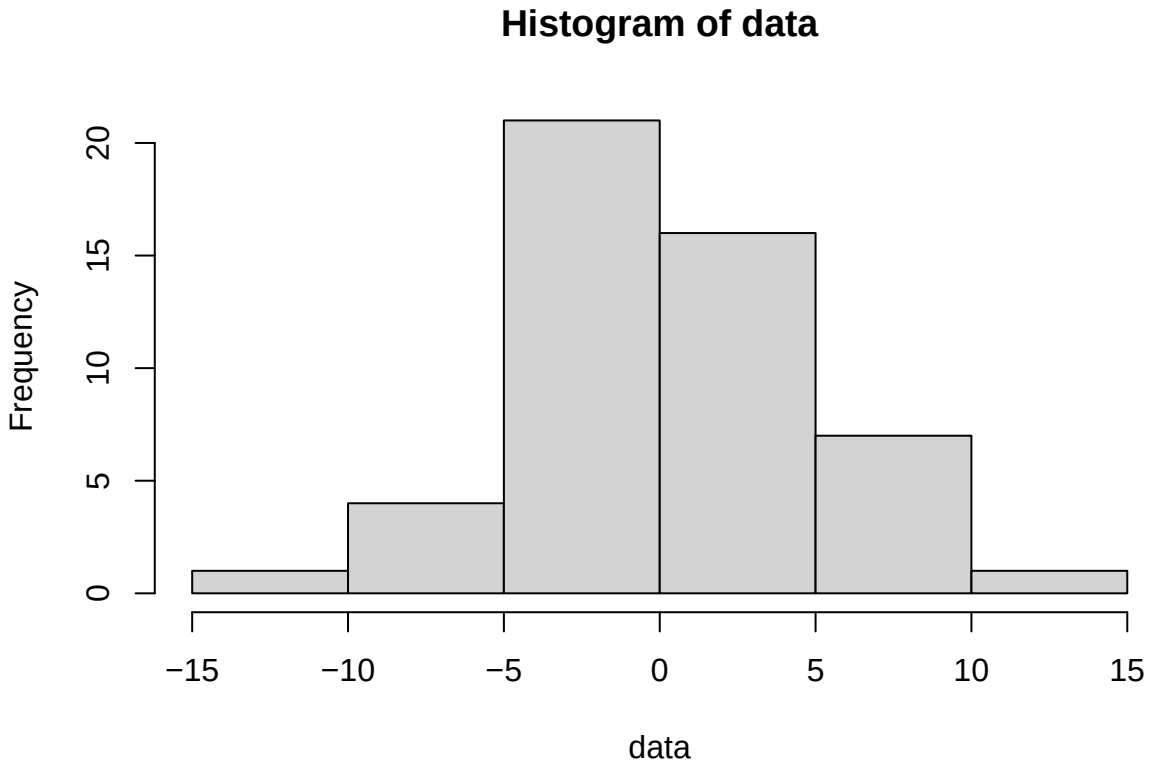
```
# using barplot() S3 Method for class `formula`
# define some data
data <- data.frame(a = c(72,41,54,36), b=c('East','West','North','South'))

# generate plot
barplot(a ~ b,
        data = data,
        col="blue",
        ylab="# of Regional Managers")
```



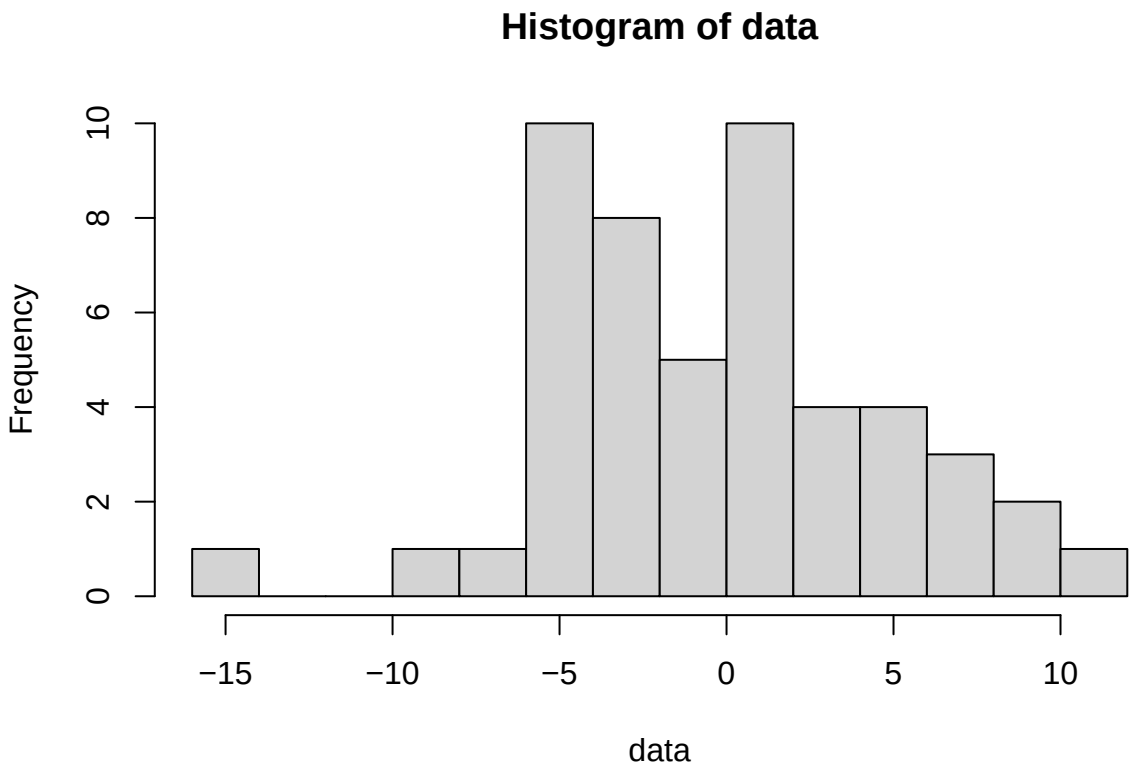
```r
bp <- barplot(a ~ b,
        data = data,
        col="blue",
        ylab="# of Regional Managers")
```

Practice Using Built-In Functions

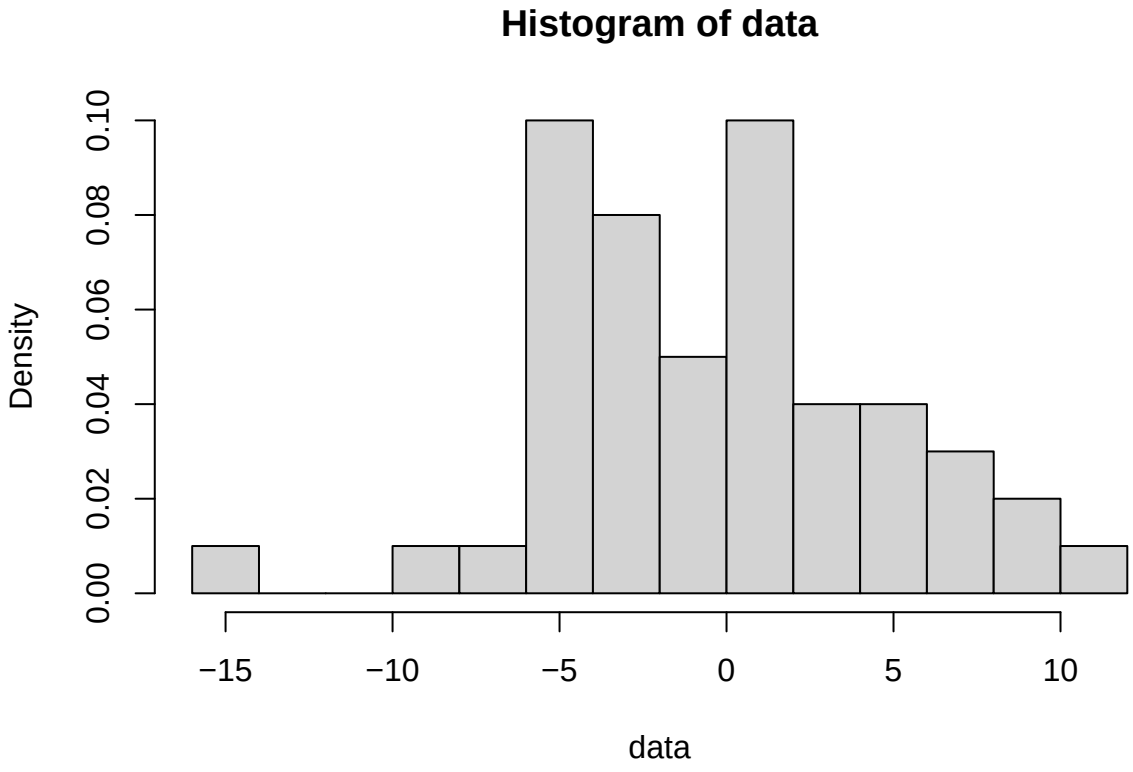
```r
# Make some data to plot. Uses rnorm() to generate 50 values drawn from a normal distribution with a me
data <- rnorm(50, mean=0, sd=4.25)

# You can use the function built in function hist() to create a histogram of data
# Use ?hist() in the Console to learn how to call the function

# What are the required parameters? Call the function using only the required parameter
hist(data)
```




```r
# add a breaks statement to increase the number of bars (bins)
# hint: count the bars in the last plot and increase it
hist(data, breaks=15)
```




```r
# set the freq argument to TRUE and then FALSE and look at the difference
hist(data, breaks=15, freq=TRUE)
hist(data, breaks=15, freq=FALSE)
```



Packages

The Base R packages provide a lot of functionality and they will take you far. However, the beauty of open source languages is that creative, capable people like you can create packages and provide them to you. The ability to expand R's functionality is limitless and free.

Packages:

- must be installed initially
- once installed, they must be loaded using the `library()` function
  - the function will stay loaded as long as the R-session is active and not closed.
  - if the R session is closed (you exit out of R), you will need to run the `library()` command again

Let's explore a package that will be useful for importing data: `openxlsx2`.

For this we will work in the Package's tab of the Output pane. Do the following:

1. In the output pane, click on the Packages tab.
2. Click on Install
3. Type the name of the package: "openxlxs2" and hit enter

This will installs the package in your environment.

Next:

1. Scroll through the list of packages in the Packages pane and look for "openxlsx2".
2. Start typing "openxlsx2" in the search bar to find it quicker.
3. Is there a check in the box to the left of the name?
   - Yes: the library is already loaded.
     - Uncheck it, so we can load it
   - No: then we need to load it before we use it.

```
library(openxlsx2)

# load NCSU Mascot data using openxlsx2
# Because we are in a notebook it sees the current working directory as "notebooks"
# To use a relative file name, we use "../" to "go" up a level and then "/data/"
# points to documents in the data directory. Once that is entered if we hit Tab we will get a list of f

# Load data using read_xlsx() function
mascot.data <- read_xlsx("../data/NCSU_Mascots_v1.xlsx")
# print the first 6 rows of the data using the head() function
print(head(mascot.data))
```

```
##                        Name Species            Breed Prior Position
## 2 Wallace Whitfield Riddick   Human             <NA>           <NA>
## 3                      Tige     Dog  English Bulldog           <NA>
## 4                      Togo     Dog American Bulldog           <NA>
## 5                     State    Wolf      Timber Wolf           <NA>
## 6                   Lobo I     Wolf      Timber Wolf            Zoo
## 7                  Lobo II     Wolf      Timber Wolf           <NA>
##     Prior Employer Interview Date Hire Year  Hire Date Termination Year
## 2             <NA>     1899-02-17      1899 1899-02-23               NA
## 3             <NA>     1910-02-16      1910 1910-02-17               NA
## 4             <NA>     1910-06-28      1910 1910-07-01               NA
## 5             <NA>     1940-05-02      1940 1940-05-03             1946
## 6 Philadelphia Zoo     1959-08-26      1959 1959-09-03             1959
## 7             <NA>     1960-02-17      1960 1960-02-23               NA
```

```
##   Termination Date Termination Reason Post Employment Position Nick Name
## 2             <NA>               <NA>                      <NA>      <NA>
## 3             <NA>               <NA>                      <NA>      <NA>
## 4             <NA>               <NA>                      <NA>      <NA>
## 5       1940-05-17            New Job            Traveling Zoo      <NA>
## 6       1959-05-29        Passed Away                      <NA>      <NA>
## 7             <NA>            Escaped                Art Thief      <NA>
##          Temperment   Fur Color  Age Height Weight Offer Sent Starting Salary
## 2           patient       cream  5.0   34.5  40.00          1            9481
## 3             sweet       brown 10.9   14.2  51.00          1           10130
## 4             sweet       brown 10.4   24.4  90.00          1            9572
## 5        bewildered multi-color  9.8   29.8 112.50          1            9921
## 6             quiet        gray 10.3   27.2 100.15          1            9509
## 7 devilishly clever multi-color  7.4   31.0 120.73          1           10424
##   Biter Hired
## 2     0     1
## 3     0     1
## 4     0     1
## 5     0     1
## 6     0     1
## 7     0     1
```

While we are using this package, let's practice writing data to an excel file.

```
# Let's write out the first 20 rows of the mascot data to an .xlsx file. Save the file in the output fo
write_xlsx(mascot.data[1:20, ], "../output/mascot_20.xlsx")

# Open the data and compare it to "../data/NCSU_Mascots_v1.xlsx"
```

User-defined Functions

Functions should:

- perform a discrete task or set of tasks
- be written to be reusable

Whether you are just starting our or whether you are a seasoned programming, spending time with paper and pencil before programming is a good idea. At a minimum, consider:

- what will the function do?
- what information (parameters) does it need to do it
- what parameters will be required vs. optional
  - it's ok to make everything required
- what will the function return?

Let's work through a couple of simple examples:

The mean of a set of numbers is $(x_1 + x_2 + ... + x_n)/N$, where $x_1, x_2, ..., x_n$ are a vector of numbers and N is the number of elements in the vector. Base R has a function to calculate the mean. We are going to make our own and use the Base R `mean()` function to check our work.

Let's consider the following before we start writing code:

1. What are the steps we need to take to calculate a mean? Think in terms of using a calculator or pen and paper.
2. What information do we need to pass to the function to calculate a mean?

3. What information do we expect to return once we've calculated the mean?
4. What data types or structures would be appropriate?
    - for parameters
    - for return values
5. Looking at our steps would we need one of our flow control statements:
    - In the steps we articulated above, do we need to run code conditionally?
    - Are there any steps that require us to do anything a preset # of times?
    - Do we have steps that need to be repeated but we aren't sure how many?

Now, we probably have enough information to start writing our function. For readability, let's use comments to document some information about our function. Also, let's call our function **avg** since the mean function already exists.

```
# Function: avg(x, N)
# Required parameters:
#   * x: a vector of numbers
#   * N: the number of elements in the vector
# Return value: avg returns the mean of the numbers in the vector.

avg <- function(x, N){
  sumX <-  0
  for (i in 1:N) {
    sumX <- sumX + x[i]
  }
  avgX <- sumX/N
  return(avgX)
}
# Once we run this code we should see `avg` listed under "Functions" in the Environment tab of the Envi

# Let's Test our Function
# Define some vectors
vector.1 <- runif(50, min=0, max=100) # pulls 50 values from a uniform distribution
vector.2 <- rnorm(50, 0, 1) # pulls 50 numbers from a normal distributions

print(avg(vector.1, 50))
```

```
## [1] 52.75223
```

```
print(mean(vector.1))
```

```
## [1] 52.75223
```

```
print(avg(vector.2, 50))
```

```
## [1] -0.01174031
```

```
print(mean(vector.2))
```

```
## [1] -0.01174031
```

Practice Writing Your Own Function

Your turn!! Try writing a function to calculate standard deviation. The formula for **s**tandard **d**eviation is: $sd = \sqrt{\frac{(x_1-\mu)^2+(x_2-\mu)^2+...+(x_n-\mu)^2}{N-1}}$ where $x_1, x_2, ...x_n$ are our values, μ is the avg of our numbers, and N is the number of values we want to find the standard deviation for. Name your function "stndev" and test it using the R `sd()` function. Before writing code, walk through the steps listed above. You can use the `sqrt()` function to calculate the square root.

Example Solution:

1. What are the steps we need to take to calculate a standard deviation?
    - We need to:
        - Calculate the mean of the numbers.
        - For each number, we need to subtract the mean from it and then square the result
        - We need to add this value to a running sum
        - Once we have calculated the sum, we need to divide the sum be the number of values - 1
        - Finally, we need to take the square root of the value calculated in the previous step
2. What information do we need to pass to the function to calculate a standard deviation?
    - a vector of numerical values
    - the number of values passed
3. What information do we expect to return once we've calculated the mean?
    - the standard deviation
4. What data types or structures would be appropriate?
    - for parameters: a vector of numerical values
    - for return values: a number representing the standard deviation
5. Looking at our steps would we need one of our flow control statements:
    - In the steps we articulated above, do we need to run code conditionally? No.
    - Are there any steps that require us to do anything a preset # of times?
        - we can use our avg() function to calculate the mean
        - we need to subtract the mean from each of our numbers, square the result and add the result to a variable. This sounds like a `for` loop (but a `while` loop could work too)
        - we can divide the result by N-1 with a single line of code
        - we can take the square root of that value with a single line of code
    - Do we have steps that need to be repeated but we aren't sure how many? No.

```
# Make sure we've run the code defining our "avg" function
# Hint: look in the Environment Tab of the Environment Pane!
standev <- function(x, N){
  # calculate mean
  my.avg <- avg(x, N)
  # initialize our running sum
  my.sum <- 0
  for (num in x) {
    # subtract mean from our number and square it
    my.value <- (num-my.avg)**2
    # add this value to our running sum
    my.sum <- my.sum + my.value
  }
  my.variance <- my.sum / (N-1)
  my.standev <- sqrt(my.variance)
  return(my.standev)
}

# test your function
# Define some vectors
vector.1 <- runif(50, min=0, max=100) # pulls 50 values from a uniform distribution
vector.2 <- rnorm(50, 0, 1) # pulls 50 numbers from a normal distributions
```

```r
# compare output from your function to the R function `sd()`
print(standev(vector.1, 50))
```

```
## [1] 25.75453
```

```r
print(sd(vector.1))
```

```
## [1] 25.75453
```

```r
print(standev(vector.2, 50))
```

```
## [1] 0.7593865
```

```r
print(sd(vector.2))
```

```
## [1] 0.7593865
```