STAT 107 Outline of Class Notes

Rebecca Kurtz-Garcia

2021-09-22

Contents

Welcome **5**

1 Introduction to R **7**
1.1 Download and Install R and Rstudio 7
1.2 The **RStudio** Interface 8
1.3 Comments 13
1.4 Operators 13
1.5 Additional Resources 17

2 Introduction to R Objects **19**
2.1 Atomic Objects 19
2.2 Vectors 22
2.3 Lists 24
2.4 Matrices 25
2.5 Factors 26
2.6 Data Frames 27
2.7 Other Object Types and the Global Environment 28
2.8 Additional Resources 28

3 More on R Objects **29**
3.1 Factors 29
3.2 Lists 32

4 Working with Data Sets **39**
4.1 Getting Data Sets in Our Working Environment 39
4.2 Basic Data Manipulation 41

5 Functions **49**
5.1 Build Your Own Function 49
5.2 Lexical Scoping 51
5.3 Built-In Functions 54
5.4 Help Files 55
5.5 The … Argument 56
5.6 Additional Resources 57

6 If Statements **59**
6.1 If . . . 59
6.2 If Else . . . 60
6.3 Else If . . . 60
6.4 Ifelse . . . 61
6.5 Nested If Chains . . . 62
6.6 Additional Resources . . . 62

7 Base R Plotting **65**
7.1 Load A Big Data Set . . . 65
7.2 Histograms . . . 67
7.3 Boxplot . . . 75
7.4 Scatter Plot . . . 77
7.5 Pie Charts . . . 77

8 Packages **81**
8.1 Namespace Collisions . . . 82

9 Loops **85**
9.1 While Loop . . . 85
9.2 For Loops . . . 87
9.3 Break . . . 88
9.4 Next . . . 89
9.5 Bisection . . . 90
9.6 Nested Loops . . . 90
9.7 Additional Resources . . . 90

10 Apply Family of Functions **91**
10.1 apply() function . . . 91
10.2 lapply() function . . . 92
10.3 sapply() function . . . 94
10.4 tapply() function . . . 95

Welcome

Welcome to STAT 107! This document will contain and outline of the course notes throughout the quarter. Please see the course website for an approximate schedule. In each of the chapters there will be a list of links, resources, and videos for learning more about an individual topic. This document will be updated constantly, be sure to check here for periodic updates. In addition, this document does not serve as a substitute for in class instruction, but more as a guide for the general content we discuss. Students are still expected to attend every lecture.

Chapter 1

Introduction to R

In this chapter we introduce R and RStudio, which you'll be using throughout this course to learn how to analyze real data and come to informed conclusions. To straighten out which is which: R is the name of the programming language itself, and RStudio is a convenient interface for using R.

As the course progresses, you are encouraged to explore beyond what we discuss; a willingness to experiment will make you a much better scientist and researcher. Before we get to that stage, however, you need to build some competence in R. We begin with some of the fundamental building blocks of R and Rstudio: the interface, data types, variables, importing data, and plotting data.

R is widely used by the scientific community as a no-cost alternative to expensive commercial software packages like SPSS and MATLAB. It is both a statistical software analysis system and a programming environment for developing scientific applications. Scientists routinely make available for free R programs they have developed that might be of use to others. Hundreds of packages can be downloaded for all types of scientific computing applications. This chapter was written by the help of Dr. Robert Desharnais [2020].

1.1 Download and Install R and Rstudio

To get started, you need to download both the R and Rstudio software. Both are available for free and there are versions for *Linux*, *Mac OS X*, and *Windows*. It is suggested that you download R first and then Rstudio. R can be used without RStudio, but RStudio provides a convenient user interface and programming environment for R.

The details for downloading and installing these software packages varies depending on your computer and operating system. You may need permission

to install the software on your computer. The links below provide access to a mirror archive at UCLA for downloading R and the developer's site for downloading RStudio.

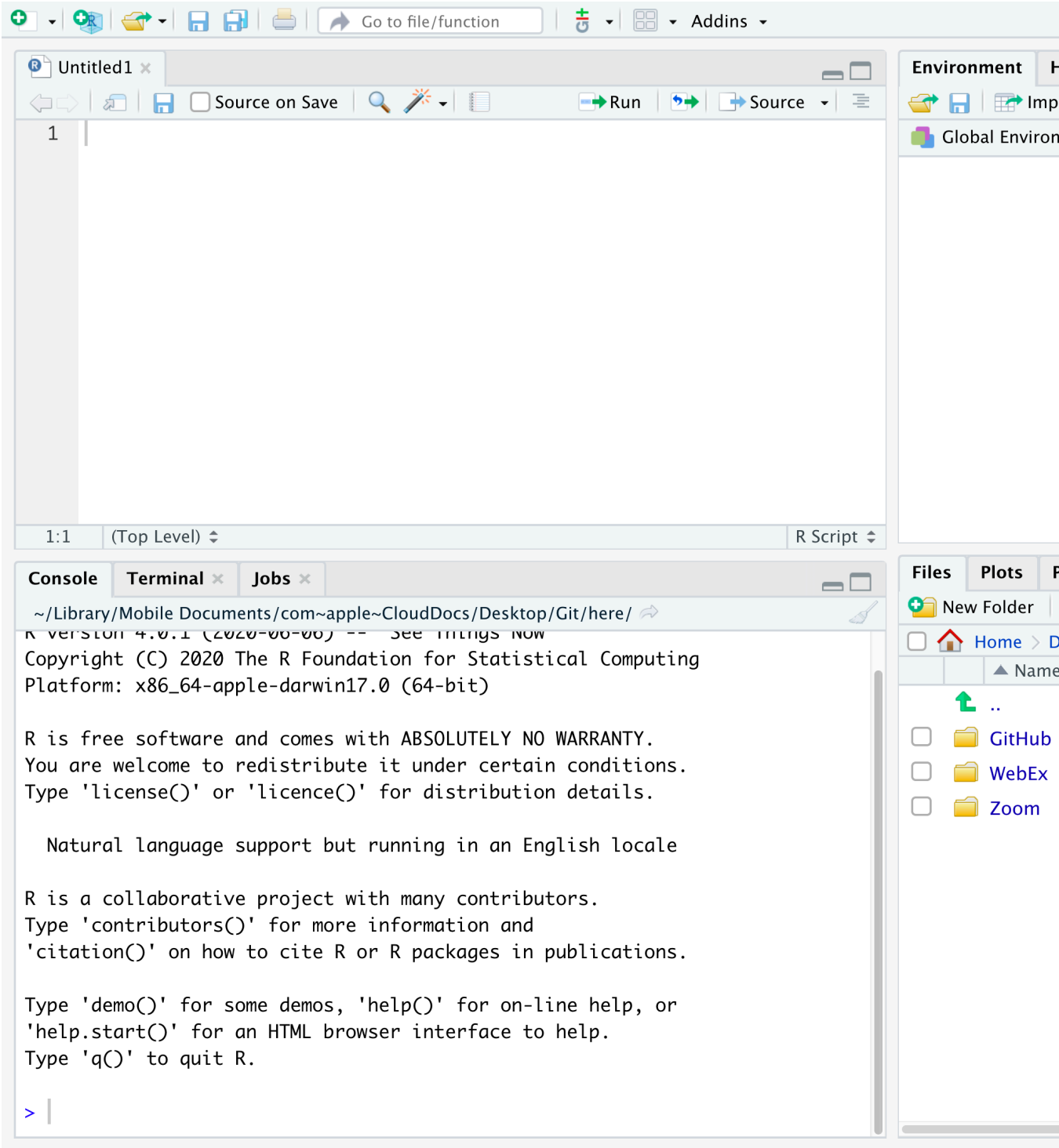
- To download R, go to the url https://ftp.osuosl.org/pub/cran. Choose the binary distribution appropriate for your computer.
  - *Windows* users will want to click on the link to "Download R for Windows" and choose "install R for the first time," then "Download R 3.5.2 for Windows."
  - *Mac OS* users will want to click "Download R for (Mac) OS X." Download the install package for version R-3.5.2 If you are using Mac OS X 10.9-10.10, install version R-3.3.3. If you are using OS X 10.6-10.8, install version R-3.2.1.
  - *Linux* users will want to click on the link to "Download R for Linux." You will need to choose the version of Linux that corresponds to your installation. Versions are available for Debian, RedHat, SUSE, and Ubuntu.

- For RStudio, use the url https://www.rstudio.com/products/rstudio/download/. Choose the binary distribution appropriate for your computer. Installers are provided for a variety of platforms.

- For additional help please see this video: Getting Started 1 | How to Download and Install RStudio

1.2 The RStudio Interface

We will begin by looking at the RStudio software interface.


Go to file/function
Addins
Untitled1
Source on Save
Run
Source
1
1:1
(Top Level)
R Script
Environment
Global Environ
Console
Terminal
Jobs
~/Library/Mobile Documents/com~apple~CloudDocs/Desktop/Git/here/
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
Files
Plots
New Folder
Home
Name
..
GitHub
WebEx
Zoom

Launch RStudio. You will see a window that looks like the figure above. There are four panels of the window:

- The pane in the bottom left is the R Command Console is where you type R commands for immediate execution.
- The pane in the upper left portion of the window is an area for editing R source code for scripts and functions and for viewing R data frame objects. New tabs will be added as new R code files and data objects are opened.
- The pane in the upper right portion of the window is an area for browsing the variables in the R workspace environment and the R command line history.
- The pane in the lower right portion of the window has several tabs. The Files tab is an area for browsing the files in the current working directory. The Plot tab is for viewing graphics produced using R commands. The Packages tab lists the R packages available. Other packages can be loaded. The Help tab provides access to the R documentation. The Viewer tab is for viewing local web content in the temporary session directory (not files on the web).

Bottom Left Pane

Let's begin with the Console. This is where you type R commands for immediate execution. Click in the Command Console, ">" symbol is the system prompt. You should see a blinking cursor that tells you the console is the current focus of keyboard input. Type:

```
1+2
```

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

The result tells you that the line begins with the first (and only) element of the result which is the number 3. You can also execute R's built-in functions (or functions you add). Type the following command.

```
exp(pi)
```

```
## [1] 23.14069
```

In R, "pi" is a special constant to represent the number and "exp" is the exponential function. The result tells you that the first (and only) element of the result is the number $e^{\pi} = 23.14069$.

Bottom Right Pane

Now let's look at the *Files* tab of the notebook at the lower right of the window. Every R session has a working directory where R looks for and saves files. It is a good practice to create a different directory for every project and make that

directory the working directory. For example, let's make a new directory called *MyDirectory*. (You can chose another name if you wish).

1) Click on the **Files** tab of the notebook. You should see a listing of files in your default working directory.
2) Click on the small button with an ellipsis image on the right side of the file path above the directory listing.
3) Navigate to the folder where you want to create the new directory and click the **OK** button.
4) Click on the **New Folder** button just below the Files tab (see right).
5) Type **MyDirectory** in the panel that opens click on the folder in the Notebook.
6) Click the **More** button to the right of the New Folder button and select the menu option **Set as Working Directory**. This new folder is now the working directory for the current R session. This menu option is a short cut for a command that was automatically entered into the R console.

Top Right Pane

Next we will look at the *R environment*, also called the *R workspace*. This is where you can see the names and other information on the variables that were created during your R session and are available for use in other commands.

In the R console type:

```
a = 29.325
b = log(a)
c = a/b
```

Look at the Environment pane. The variables a, b, and c are now part of your R work space. You can reuse those variables as part of other commands.

In the R console type:

```
v= c(a, b, c)
v
```

```
## [1] 29.325000  3.378440  8.680041
```

The variable v is a vector created using the *concatenate* function c(). (The concatenate should not be confused with the variable c that was created earlier. Functions are always followed by parentheses that contain the function arguments.) This function combines its arguments into a vector or list. Look at the Environment panel. The text num [1:3] tells us that the variable v is a vector with elements v[1], v[2], and v[3].

Top Left Pane

Now let's look at the R viewer notebook. This panel can be used to data which are data frame objects or *matrix objects* in R.

We will begin by taking advantage of a data frame object that was built into R for demonstration purposes. We will copy it into a data frame object. In the R console, type:

```
df = mtcars
```

Let's view the data. On the right side of the entry for the df object is a button we can use to view the entries of the data frame. Click on the View Button.

If your look in the notebook area in the upper left portion of the window, you can see a spreadsheet-like view of the data. This is for viewing only; you cannot edit the data. Use the scroll bars to view the data entries.

You can also list the data in the console by typing the name of the data fame object:

```
df
```

```
##                   mpg cyl  disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4        21.0   6 160.0 110 3.90 2.620 16.46  0  1   4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1   4
## Datsun 710       22.8   4 108.0  93 3.85 2.320 18.61  1  1   4
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0   3
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0   3
## Valiant          18.1   6 225.0 105 2.76 3.460 20.22  1  0   3
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0   3
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0   4
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0   4
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0   4
## Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0   4
## Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0   3
## Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0   3
## Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0   3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0   3
## Fiat 128         32.4   4  78.7  66 4.08 2.200 19.47  1  1   4
## Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1   4
## Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1   4
## Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0   3
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0   3
## AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0   3
## Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0   3
## Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0   3
```

```
## Fiat X1-9        27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The columns are labeled with the names of the variables and the rows are labeled with the names of each car. Each row represents the data values for one car; that is, each row is one observation.

1.3 Comments

Often times we will want to add a comment to our script document so we can remember special aspects later, and make the code easier to read and modify in the future. To add a comment start the comment with a # symbol. This will make the remaining characters in a line a comment and R will not try to compile these lines. Go to the script document and type the following. Highlight what you have typed and press "Run".

```
# This is a comment
2+ 2
```

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

```
2 + 3 # Comments can also start in the middle of a line.
```

```
## [1] 5
```

1.4 Operators

An operator is a symbol that tells the compiler to preform a specific task. There are several types of operators, some preform mathematical tasks, logical checks, and create new objects. We will review a few of the basic operators here. We will continue to discuss and introduce operators throughout this document.

Arithmetic Operators

R was designed for statistical applications and as a necessity it needs to preform mathematical operations efficiently and effectively. The first operators we discuss are a few of the basic arithmetic operations. These are operations similar to that of a calculator.

```
# Addition
2+ 3
```

```
## [1] 5
```

```
# Subtraction
2 - 3
```

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

```
# Multiplication
2*3
```

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

```
# Division
2/3
```

```
## [1] 0.6666667
```

```
# Exponent
2^3
```

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

Relational Operators

Relational operators are used to compare two values. When using a relational operation R will return either `TRUE` or `FALSE`.

```
# Less than
2 < 3
```

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

```
# Greater than
2 > 3
```

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

```
# Less than or equal to
2 <= 3
```

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

```
# Greater than or equal to
2>= 3
```

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

```
# Not equal to
2 != 3
```

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

```
# Equal to
2 == 3
```

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

We can use all the same operators above if our object contains more than one element. This will preform the above comparisons element by element.

```
v
```

```
## [1] 29.325000  3.378440  8.680041
```

```
v > 10
```

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

If we have two vectors of an unequal length then the checks will be preformed element-by-element but the values in the shorter vector will be *recycled*, or *repeated*.

```
w = c(10, 1)
v > w
```

```
## Warning in v > w: longer object length is not a multiple of shorter obj
## length
```

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

R evaluated the first and third element of `v` and compared it to the first element of `w`, and the second element of `v` to the second element of `w`. In this case, R returned a *warning* alerting you that it recycled elements. However, R will not always give a warning.

Logical operators

Logical operators are similar to relational operators. They are used to check "AND" and "OR" events. We have the `&` symbol which returns `TRUE` only if BOTH conditions are true. We also have the `|` symbol which returns `TRUE` if EITHER condition is true.

```
# Check if both operations are true.
(2 < 3) & (5 < 4)
```

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

```
# Check if either operation is true.
(2 < 3) | (5 < 4)
```

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

We can also negate a `TRUE` or `FALSE` value using the `!` symbol.

```
# Negate an operation
!(2<3)
```

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

Like relational operators from before, if we have more than one element the logical operations will be implemented element-by-element.

```
# AND event, compared element-by-element
(v > 10) & (4 < 5)
```

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

```
# OR event, compared elmeent-by-elment
(v > 10) | (4 < 5)
```

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

We also have the symbols `&&` and `||` which will ensure that only the first element in an object will be compared.

```
# AND event, only check the first element
(v > 10) && (4 < 5)
```

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

```
# OR event, only check the first element
(v > 10) || (4 < 5)
```

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

Assignment Operators

Assignment operators are used to assign values to a new object. There are many types of assignment operators, and they operate slightly differently. The two most common assignment operators are $=$ and $<-$. With these operators the value to the left of the operator is the name of the new object and the value on the right is what the object is now equal to.

```
x = 5
x
```

```
## [1] 5
```

```
x <- 5
x
```

```
## [1] 5
```

The majority of the time we can use these two assignment operators above interchangeably, there are some exceptions though. There are several other

assignment operators which are uncommon and should only be used by advanced users, ->, <<-, and ->>.

1.5 Additional Resources

- Chapter 1 of “CRAN Intro-to-R Manual”
- Videos:
  - Getting Started 1 | How to Download and Install RStudio
  - Getting Started 2 | Rstudio Introduction cont’d, More Tabs Explained

Chapter 2

Introduction to R Objects

At its core, R is an objected-oriented computational and programming environment. Everything in R is an object belonging to a certain *class*. In this chapter we begin by discussing **atomic** objects, these are the core fundamental objects used in R. Next we discuss vectors, lists, and matrices. These objects are among the most common types that are used. Lastly we discuss factors and data frames. Factors and data frames are known as dynamic objects, and are more complex. All of these objects have special properties and a multitude of features that we can use. We discuss some of their key properties here, but will continue exploring and learning about their features, and introducing more object types throughout the course.

2.1 Atomic Objects

R can represent different types of data. The types include `numeric`, `integer`, `complex`, `logical`, `character`, and `raw`. These are the basic fundamental objects we can use in R. For our class we will not need the `complex` type which stores complex numbers, and in practice `raw` is rarely used. We will concentrate on the remaining four types. Unlike other object-oriented languages we do not need to specify what type of object we are creating when we create it. Instead, R guesses the type of object you are creating. To check the object type we can use the `class()` function.

Numeric

Numeric objects are perhaps the most common. These are objects which contain a real number, that is, a number which can contain a decimal value. These objects are comparable to `doubles` in `C`.

```
a = 17.45
a
```

```
## [1] 17.45
```

```
class(a)
```

```
## [1] "numeric"
```

```
b = 5
b
```

```
## [1] 5
```

```
class(b)
```

```
## [1] "numeric"
```

Both the variables `a` and `b` are `numeric` objects. When you type a number R will default to treating it as a `numeric` object which allows decimals.

Integer

We can also create numeric objects which are specifically made to store integer values. We can do this using the `as.integer()` function.

```
a = as.integer(a)
a
```

```
## [1] 17
```

```
class(a)
```

```
## [1] "integer"
```

```
b = as.integer(b)
b
```

```
## [1] 5
```

```
class(b)
```

```
## [1] "integer"
```

Logical

Logical values are either `TRUE` or `FALSE` and are created by using logical and relational operators. In other words, they are created by using statements that compare variables. There are several ways to do logical statements as we saw in Section 1.4.

```
b = 5
n = (10<11)
n
```

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

```
class(n)
```

```
## [1] "logical"
```

We can also assign a value as `TRUE` or `FALSE` manually by setting it equal to `TRUE` or `FALSE`, or by using `T` or `F`.

```
c = T
c
```

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

```
class(c)
```

```
## [1] "logical"
```

Character

`Character` values are text. They are often used as data values and labels.

```
first = "George"
first
```

```
## [1] "George"
```

```
class(first)
```

```
## [1] "character"
```

```
last = "Washington"
last
```

```
## [1] "Washington"
```

```
class(last)
```

```
## [1] "character"
```

There are several functions that can operate on character strings.

```
full = paste(first, last)
full
```

```
## [1] "George Washington"
```

```
nchar(full)
```

```
## [1] 17
```

```
tolower(full)
```

```
## [1] "george washington"
```

```
toupper(full)
```

```
## [1] "GEORGE WASHINGTON"
```

The function `paste()` concatenates two or more character strings with a separator, which is a space by default. The function `nchar()` returns the number of characters in a string. The functions `tolower()` and `toupper()` changes any upper case characters to lower case and vice-versa.

2.2 Vectors

All the objects we have created this far are single element *vectors*. R is a vectorized language, meaning most of the procedures, functions, and operations have been optimized to work with vectors. It is typically advantageous to utilize this feature. **A vector is a collection of values of the same data type**. We can use the concatenate function, `c()`, to create vectors, and to make a vector larger.

```
v1 = c(19, 390.3, pi, -32.1)
v1
```

```
## [1]   19.000000  390.300000    3.141593  -
32.100000
```

```
class(v1)
```

```
## [1] "numeric"
```

```
v2 = c(1.1, 6, -9.4, 32.1)
v2
```

```
## [1]  1.1  6.0 -9.4 32.1
```

```
class(v2)
```

```
## [1] "numeric"
```

If we try to create a vector with a mix of classes R will convert all the objects to be the same class. In general, it is easiest to convert objects into a `character` but hard to convert `character` into something else. Be cautious when mixing data types and vectors because you will not be notified if objects are converted, and they may not be converted to the class you intended.

```
v3 = c(v1, first)
class(v3)
```

```
## [1] "character"
```

```
v4 = c(first, last)
class(v4)
```

```
## [1] "character"
```

The `length()` function can be used to obtain the number of elements in a vector.

```
length(v1)
```

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

Vectors can be used in arithmetic computations. If the two vectors are of the same length, the computations are performed element-by-element.

```
v1 + v2
```

```
## [1]  20.100000 396.300000  -6.258407   0.000000
```

```
v1 * v2
```

```
## [1]     20.90000   2341.80000    -29.53097 -1030.41000
```

Single numbers (scalars) will operate on all the vector elements in an expression.

```
5*v1
```

```
## [1]     95.00000  1951.50000     15.70796  -160.50000
```

```
v1/3
```

```
## [1]     6.333333  130.100000     1.047198  -10.700000
```

Individual elements of a vector can be obtained using an index in square brackets. A negative index removes that element from the vector. The `v2[-1]` is the vector `v2` with the first element removed. The concatenate function can be used to obtain two or more elements of a vector in any desired order. Here `v1[c(3,2)]` returns the third and second elements of the vector `v1`.

```
v1[3]
```

```
## [1] 3.141593
```

```
v2[-1]
```

```
## [1]  6.0 -9.4 32.1
```

```
v3[c(3,2)]
```

```
## [1] "3.14159265358979" "390.3"
```

2.3 Lists

Lists are thought of as a vector with a variety of classes. A list is made up of elements, and each element can be of a different class.

```
lst = list(4, v4, v2)
lst
```

```
## [[1]]
## [1] 4
##
## [[2]]
## [1] "George"     "Washington"
##
## [[3]]
## [1]  1.1  6.0 -9.4 32.1
```

```
class(lst)
```

```
## [1] "list"
```

We can observe the class of each element in the list by using the `str()` function.

```
str(lst)
```

```
## List of 3
##  $ : num 4
##  $ : chr [1:2] "George" "Washington"
##  $ : num [1:4] 1.1 6 -9.4 32.1
```

The above output tells us we have a list of three objects. The first object is a numeric vector with one element, the second object is a character vector with two elements, and third object is a numeric vector with four elements. We can subset elements in a list using double brackets we `[[]]`. Inside these square brakets we state the element we would like to obtain.

```
lst[[1]]
```

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

```
class(lst[[1]])
```

```
## [1] "numeric"
```

To determine how long our list is we can use the `length()` function.

```
length(lst)
```

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

2.4 Matrices

A matrix is a two dimensional array of data of **the same type**. The matrix function, `matrix()`, can be used to create a new matrix.

```
m = matrix(c(1, 9, 2, 0, 5, 7, 3, 8, 4),
           nrow=3, ncol=3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    3
## [2,]    9    5    8
## [3,]    2    7    4
```

R labels the rows and columns for us in the output. The matrix is filled column-by-column using the elements of the vector created by the concatenate function. As with vectors, matrices can be used in arithmetic operations with scalars and other matrices of the same size.

```
m2 = m/2
m2
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  0.0  1.5
## [2,]  4.5  2.5  4.0
## [3,]  1.0  3.5  2.0
```

```
m *m2
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  0.0  4.5
## [2,] 40.5 12.5 32.0
## [3,]  2.0 24.5  8.0
```

Indices can be used to obtain the elements of a matrix, but now we must consider both the row and column.

```
m[2,2]
```

```
## [1] 5
```

```
m[c(1,3), c(1,3)]
```

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

```
m[2,]
```

```
## [1] 9 5 8
```

```
m[,3]
```

```
## [1] 3 8 4
```

Some functions are particularly useful when using matrices. For instance, `t()`, `dim()`, and `c()`. The transpose function, `t()`, switches the column and rows of a matrix. The dimension function, `dim()`, returns the dimensions (number of rows, columns) of a matrix. The concatenate function, `c()`, turns a matrix into a vector by concatenating the columns of the matrix.

```
# Dimensions (row, column)
dim(m)
```

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

```
# Transpose
t(m)
```

```
##      [,1] [,2] [,3]
## [1,]    1    9    2
## [2,]    0    5    7
## [3,]    3    8    4
```

```
# Convert to vector
c(m)
```

```
## [1] 1 9 2 0 5 7 3 8 4
```

2.5 Factors

Factors are useful for categorical data. Factors differ from character objects in that a character object is a string of characters or symbols placed in a specific order. For example, the object `first = "George"` is a character object with six elements. In contrast, the collection of values “George” would instead represent a distinct value, or *level*, for a factor object. We can create a factor object using the `factor()` function.

```
colors = c("red", "blue", "red", "red", "blue")
colors = factor(colors)
colors
```

```
## [1] red  blue red  red  blue
## Levels: blue red
```

```
class(colors)
```

```
## [1] "factor"
```

Here the unique elements in the factor are called “levels”. There are only two levels `red` and `blue`, and there are five elements in our factor object. We can index factors in the same way we index a list.

```
# Return first element in the factor object
colors[1]
```

```
## [1] red
## Levels: blue red
```

2.6 Data Frames

Like a matrix, a data frame is a rectangular array of values where each column is a vector, However, unlike a matrix, the columns can be different data types.

We can create a set of vectors of the same length and use the `data.frame()` function to make a data frame object.

```
age = c(1, 8, 10, 30, 31)
gender = c("Female", "Female", "Male","Female","Male")
married = c(FALSE, FALSE, FALSE, TRUE, TRUE)
simpsons = data.frame(age, gender, married)
simpsons
```

```
##   age gender married
## 1   1 Female   FALSE
## 2   8 Female   FALSE
## 3  10   Male   FALSE
## 4  30 Female    TRUE
## 5  31   Male    TRUE
```

```
class(simpsons)
```

```
## [1] "data.frame"
```

Data frames can be indexed using the same techniques as was done with matrices.

2.7 Other Object Types and the Global Environment

There are more objects then what we have discussed above. For example, many of the advanced functions create specific objects generated by that specific function. There are hundreds, and possibly thousands, of such objects. These objects generally are special cases of lists, factors, and other various types of objects that we have defined in this section. The objects we have described here are the building blocks of most values we will be working with. Functions like `class()` and `length()` are also considered as objects, but are of a different type. We discuss functions in more detail in section 5.

There are also built-in, or special objects in R. For example, the object `pi` is an object already defined. These built-in values and functions can be written over, but that is not advised.

```
pi
```

```
## [1] 3.141593
```

Every time we create an object we see that the Global Environment tab in the top right pane updates. The object we have created is now listed in the Global Environment. This is a collection of all *user created* objects in R, that R knows about, and that R can easily call. Built-in objects, such as `pi`, will not be listed here.

2.8 Additional Resources

- Chapters 2, 3, 4.1, 4.3, 5.1-5.3, 6 of CRAN Intro-to-R Manual
- Videos:
  - Variables 1 | Types and Assignments
  - Variables 2 | Nameing Conventions and Best Practices
  - Vectors 1 | Introduction
  - Vectors 2 | Subsetting and Modifying
  - Vectors 3 | Vectorized Functions - Logical Comparisons
  - Matrices 1 | Introduction
  - Matrices 2 | Accessing Rows and Columns

Chapter 3

More on R Objects

Factors and lists have especially unique properties that are often utilized. In this chapter we take special care to discuss these properties.

3.1 Factors

In real-world problems, you often encounter data that can be classified in categories. For example, suppose a survey was conducted of a group of seven individuals, who were asked to identify their hair color and if they own a pet.

```
name = c("Amy", "Bob", "Eve", "Kim", "Max", "Ray", "Sam")
hair = c("Blonde", "Black", "Black", "Red", "Blonde", "Brown", "Black
own_pets = c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE)

catagorical = data.frame(name, hair, own_pets)
colnames(catagorical) = c("Name", "Hair Color", "Own Pets")

catagorical
```

```
##   Name Hair Color Own Pets
## 1  Amy     Blonde     TRUE
## 2  Bob      Black    FALSE
## 3  Eve      Black     TRUE
## 4  Kim        Red     TRUE
## 5  Max     Blonde    FALSE
## 6  Ray      Brown    FALSE
## 7  Sam      Black    FALSE
```

Here, the hair color and pet ownership are examples of categorical data. For the hair color variable we will typically want to store it as a factor, as opposed

to a character vector. The different values that the factor can take are called levels. In R, you can create a factor with the `factor()` function.

```
f = factor(hair)
f
```

```
## [1] Blonde Black  Black  Red   Blonde Brown  Black
## Levels: Black Blonde Brown Red
```

A factor looks like a vector, but it has special properties. Levels are one of them. Notice that when you print the factor, R displays the distinct levels below the factor. R keeps track of all the possible values in a vector, and each value is called a level of the associated factor.The `levels()` function shows all the levels from a factor.

```
levels(f)
```

```
## [1] "Black"  "Blonde" "Brown"  "Red"
```

If your vector contains only a subset of all the possible levels, then R will have an incomplete picture of the possible levels. Consider the following example of a vector consisting of directions:

```
directions = c("North", "West", "North", "East", "North", "Wes
f = factor(directions)
f
```

```
## [1] North West  North East  North West  East
## Levels: East North West
```

Notice that the levels of your new factor do not contain the value “South”. So, R thinks that North, West, and East are the only possible levels. However, in practice, it makes sense to have all the possible directions as levels of your factor. To add all the possible levels explicitly, you specify the `levels` argument of the function `factor()`.

```
directions = c("North", "West", "North", "East", "North", "Wes
f = factor(directions,
           levels = c("North", "East", "South", "West"))
f
```

```
## [1] North West  North East  North West  East
## Levels: North East South West
```

R lets you assign abbreviated names for the levels. You can do this by specifying the `labels` argument of `factor()`.

```
directions = c("North", "West", "South", "East", "West", "Nort
f = factor(directions,
           levels = c("North", "East", "South", "West"),
           labels = c("N", "E", "S", "W"))
```

```
f
```

```
## [1] N W S E W N S
## Levels: N E S W
```

Sometimes data has some kind of natural order between elements. For example, sports analysts use a three-point scale to determine how well a sports team is competing:

loss < tie < win.

In market research, it's very common to use a five point scale to measure perceptions:

strongly disagree < disagree < neutral < agree < strongly agree.

Such kind of data that is possible to place in order or scale is known as **ordinal data**. We can store ordinal data as an ordered factor. To create an ordered factor, use the `factor()` function with the argument `ordered=TRUE`.

```
record = c("win", "tie", "loss", "tie", "loss", "win", "win")
f = factor(record,
           ordered = TRUE)
f
```

```
## [1] win  tie  loss tie  loss win  win
## Levels: loss < tie < win
```

You can also reverse the order of levels using the `rev()` function.

```
record = c("win", "tie", "loss", "tie", "loss", "win", "win")
f = factor(record,
           ordered = TRUE,
           levels = rev(levels(f)))
f
```

```
## [1] win  tie  loss tie  loss win  win
## Levels: win < tie < loss
```

If you have no observations in one of the levels, you can drop it using the `droplevels()` function.

```
record = c("win", "loss", "loss", "win", "loss", "win")
f = factor(record,
           levels = c("loss", "tie", "win"))

f
```

```
## [1] win  loss loss win  loss win
## Levels: loss tie win
```

```
droplevels(f)
```

```
## [1] win  loss loss win  loss win
## Levels: loss win
```

The `summary()` function will give you a quick overview of the contents of a factor.

```
f = factor(hair)
f
```

```
## [1] Blonde Black  Black  Red   Blonde Brown  Black
## Levels: Black Blonde Brown Red
```

```
summary(f)
```

```
##  Black Blonde  Brown    Red
##      3      2      1      1
```

The function `table()` tabulates observations.

```
table(f)
```

```
## f
##  Black Blonde  Brown    Red
##      3      2      1      1
```

3.2 Lists

A *list* is an array of objects. Unlike vectors and matrices, the elements in a list can belong to different classes. Lists are useful for packaging together a set of related objects. We can create a list of objects in our environment by using the `list()` function.

```
lst = list(1, 2, 3)

# A list of characters
lst = list("red", "green", "blue")

# A list of mixed datatypes
lst = list(1, "abc", 1.23, TRUE)
```

The best way to understand the contents of a list is to use the structure function `str()`. It provides a compact display of the internal structure of a list.

```
lst = list(1, "abc", 1.23, TRUE)
str(lst)
```

```
## List of 4
```

```
##   $ : num 1
##   $ : chr "abc"
##   $ : num 1.23
##   $ : logi TRUE
```

A list can contain sublists, which in turn can contain sublists themselves, and so on. This is known as *nested list* or *recursive vectors*.

```
lst = list(1, 3, "abc", list("a","b","c"), TRUE)
str(lst)
```

```
## List of 5
##   $ : num 1
##   $ : num 3
##   $ : chr "abc"
##   $ :List of 3
##    ..$ : chr "a"
##    ..$ : chr "b"
##    ..$ : chr "c"
##   $ : logi TRUE
```

There are two ways to extract elements from a list:

- Using `[[]]` gives you the element itself.
- Using `[]` gives you a list with the selected elements

You can use `[]` to extract either a single element or multiple elements from a list. However, the result will always be a list.

```
# extract 2nd element
lst[2]
```

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

```
# extract 5th element
lst[5]
```

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

```
# select 1st, 3rd and 5th element
lst[c(1,3,5)]
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "abc"
##
```

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

```
# exclude 1st, 3rd and 5th element
lst[c(-1,-3,-5)]
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [[2]][[1]]
## [1] "a"
##
## [[2]][[2]]
## [1] "b"
##
## [[2]][[3]]
## [1] "c"
```

You can use `[[]]` to extract only a single element from a list. Unlike `[]`, `[[]]` gives you the element itself.

```
# extract 2nd element
lst[[2]]
```

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

```
# extract 5th element
lst[[5]]
```

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

You can't use logical vectors or negative numbers as indices when using `[[]]`. The difference between `[]` and `[[]]` is really important for lists, because `[[]]` returns the element itself while `[]` returns a list with the selected elements. The difference becomes clear when we inspect the structure of the output – one is a character and the other one is a list.

```
lst = list("a","b","c","d","e","f")

class(lst[[1]])
```

```
## [1] "character"
```

```
class(lst[1])
```

```
## [1] "list"
```

Each list element can have a name. You can access individual element by specifying its name in double square brackets `[[]]` or use `$` operator.

```
months = list(JAN=1, FEB=2, MAR=3, APR=4)

# extract element by its name
months[["MAR"]]
```

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

```
# same as above but using the $ operator
months$MAR
```

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

```
# extract multiple elements
months[c("JAN","APR")]
```

```
## $JAN
## [1] 1
##
## $APR
## [1] 4
```

You can access individual items in a nested list by using the combination of `[[]]` or `$` operator and the `[]` operator.

```
lst = list(item1 = 3.14,
           item2 = list(item2a = 5:10,
                        item2b = c("a","b","c")))

# preserve the output as a list
lst[[2]][1]
```

```
## $item2a
## [1]  5  6  7  8  9 10
```

```
# same as above but simplify the output
lst[[2]][[1]]
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above with names
lst[["item2"]][["item2a"]]
```

```
## [1]  5  6  7  8  9 10
```

```
# same as above with $ operator
lst$item2$item2a
```

```
## [1]  5  6  7  8  9 10
```

```
# extract individual element
lst[[2]][[2]][3]
```

```
## [1] "c"
```

Modifying a list element is pretty straightforward. You use either the `[[]]` or the `$` to access that element, and simply assign a new value.

```
# Modify 3rd list element
lst = list("a","b","c","d","e","f")
lst[[3]] = 1
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : num 1
##  $ : chr "d"
##  $ : chr "e"
##  $ : chr "f"
```

You can modify components using `[]` as well, but you have to assign a list of components.

```
# Modify 3rd list element using []
lst = list("a","b","c","d","e","f")
lst[3] = list(1)
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : num 1
##  $ : chr "d"
##  $ : chr "e"
##  $ : chr "f"
```

Using `[]` allows you to modify more than one component at once.

```
# Modify first three list elements
lst = list("a","b","c","d","e","f")
lst[1:3] = list(1,2,3)
str(lst)
```

```
## List of 6
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : chr "d"
```

```
##   $ : chr "e"
##   $ : chr "f"
```

You can use same method for modifying elements and adding new one. If the element is already present in the list, it is updated else, a new element is added to the list.

```
# Add elements to a list
lst = list(1, 2, 3)
lst[[4]] = 4
str(lst)
```

```
## List of 4
##   $ : num 1
##   $ : num 2
##   $ : num 3
##   $ : num 4
```

By using `append()` method you can append one or more elements to the list.

```
# Add more than one element to a list
lst = list(1, 2, 3)
lst = append(lst,c("a","b","c"))
str(lst)
```

```
## List of 6
##   $ : num 1
##   $ : num 2
##   $ : num 3
##   $ : chr "a"
##   $ : chr "b"
##   $ : chr "c"
```

By using a logical vector, you can remove list elements based on the condition.

```
# Remove all negative list elements
lst = list(-4,-3,-2,-1,0,1,2,3,4)
lst[lst <= 0] = NULL
str(lst)
```

```
## List of 4
##   $ : num 1
##   $ : num 2
##   $ : num 3
##   $ : num 4
```

The `c()` does a lot more than just creating vectors. It can be used to combine lists into a new list as well.

```
lst1 = list("a","b","c")
lst2 = list(1,2,3)
lst = c(lst1, lst2)
str(lst)
```

```
## List of 6
##  $ : chr "a"
##  $ : chr "b"
##  $ : chr "c"
##  $ : num 1
##  $ : num 2
##  $ : num 3
```

Basic statistical functions work on vectors but not on lists. For example, you cannot directly compute the mean of list of numbers. In that case, you have to flatten the list into a vector using `unlist()` first and then compute the mean of the result.

```
lst = list(5, 10, 15, 20, 25)
mean(unlist(lst))
```

```
## [1] 15
```

To find the length of a list, use `length()` function.

```
length(lst)
```

```
## [1] 5
```

Chapter 4

Working with Data Sets

In this section we discuss different methods for loading data sets into our R session. There are many different files we can create and import. We will focus our attention on loading csv files because they tend to be easier to import, and they are one of the more typical file types that are used. In the second half of the this chapter we introduce basic data manipulation strategies and helpful functions when working with data sets.

4.1 Getting Data Sets in Our Working Environment

Built-In Data

As discussed last week, there are built in objects which are not loaded into the global environment, but can be called upon at any time. For example `pi` returns the value 3.1415927. Similarly, there are built in data sets that are ready to be used and loaded at a moments notice.

1) To see a list of built in data sets type in the console:

```
data()
```

2) These data sets can be used even if they are not listed in the global environment. For example, if you would like to load the data set `cars` in the global environment, run the following command:

```
data("cars")
```

Importing From Your Computer

Although built-in data sets are convient, most the time we need to load our own datasets. We load our own data sets by using a function specifically designed

for the file type of interest. This function usually uses the file path location as an argument. This can be done in many different ways; however, we will only go over two.

Option 1

1) Download the file `InsectData.csv` from **ELearn**. Save this file in a spot in your computer you will remember.

2) In the **Environment** window (upper left window), click on the **Import Dataset** button. A drop down menu will appear. Select the **From Text (base)…** option. Find the file `InsectData.csv` and select it.

3) A pop up menu will appear giving you options for loading in the file, and showing a preview of what the file will look like once loaded. Select the appropriate options and click **Import**.

4) A new line of code has generated in the console which will read the data into your current environment. Copy and paste this into your R script document if you would like to save this line of code for later. You will have to reload this file into your environment each time you start a new R session and would like to use this file.

Option 2

1) Download the file `InsectData.csv` from **ELearn**. Save this file in a spot in your computer you will remember.

2) In the lower right hand window select the **File** tab. Now search for the file which you have saved `InsectData.csv`.

3) Click on the file `InsectData.csv` in order to see a dropdown menu. Select **Import Dataset…**

4) A window will appear which will give you options and a preview of your file. Select appropriate options if needed then click **Import**.

5) A new line of code has generated in the console which will read the data into your current environment. Copy and paste this into your R script document if you would like to save this line of code for later. You will have to reload this file into your environment each time you start a new R session and would like to use this file.

Import From Online

We can also download data sets from online in a variety of different ways. Below is one option. With this method we are using the same `InsectData.csv` file, but it has been posted online. We feed the url of where the data set has been posted into the `read.csv()` function in order to open the file.

```
the_url = "https://raw.githubusercontent.com/rpkgarcia/LearnRBook/mai
the_data = read.csv(the_url)
```

4.2 Basic Data Manipulation

Lets recall a few useful things about data frames. As we learned already, data sets are contained in an object called a data frame. One can view this as a specialized table or matrix of rows and columns, where each column is a data variable, such as height or age, and each row is a single observation. All of the values within a column must be the same data type (numeric,factor, logical, etc.). Data frames can be created or called within R, imported from text or spreadsheet files, or imported from the web.

```
group = c("G1", "G2", "G1", "G1", "G2")
age = c(35, 30, 31, 28, 40)
height = c(65, 70, 60, 72, 68)
pets = c(TRUE, TRUE, FALSE, FALSE, TRUE)

mydata = data.frame(group, age, height, pets)
mydata
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 2    G2  30     70  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
## 5    G2  40     68  TRUE
```

We have various R functions that help provide information about the data frame.

```
# How many rows
nrow(mydata)
```

```
## [1] 5
```

```
# How many columns
ncol(mydata)
```

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

```
# Rows, Columns
dim(mydata)
```

```
## [1] 5 4
```

```
# Colnames
colnames(mydata)
```

```
## [1] "group"  "age"    "height" "pets"
```

The `summary()` function is a powerful command that gives you some summary statistics about the variales in the data frame.

```
summary(mydata)
```

```
##    group                age            height     pets
##  Length:5           Min.   :28.0   Min.   :60   Mode :logical
##  Class :character   1st Qu.:30.0   1st Qu.:65   FALSE:2
##  Mode  :character   Median :31.0   Median :68   TRUE :3
##                     Mean   :32.8   Mean   :67
##                     3rd Qu.:35.0   3rd Qu.:70
##                     Max.   :40.0   Max.   :72
```

The summary statistics are listed below the names of the variables. Since pets is a logical variable, R gives you the frequencies of each unique value. In this example there are three values of `TRUE` and two values of `FALSE`. Since age and weight are numeric, R computes and returns the minimum, 1st quartile (25th percentile), median, mean, 3rd quartile (75th percentile), and maximum values. If you have many data values, this is a quick way to get a feel for how the data are distributed.

The `table()` function can also be used to cross-tabulate categorical data. Let's create a frequency table for the different groups.

```
table(mydata$group)
```

```
##
## G1 G2
##  3  2
```

We can also create a frequency table of pet status for both groups.

```
table(mydata$group, mydata$pets)
```

```
##
##       FALSE TRUE
##    G1     2    1
##    G2     0    2
```

Indexing

As with matrices, square brackets followed by the row, column can be used to return a specific data value. For example, to get the datum in the 3rd row, 2nd column.

```
mydata[3,2]
```

```
## [1] 31
```

If you leave out the number before or after the comma, the entire column or row is returned.

```
mydata[3,]
```

```
##   group age height  pets
## 3    G1  31     60 FALSE
```

```
mydata[,2]
```

```
## [1] 35 30 31 28 40
```

For data frame you can also access variables by name instead of column number. Use a dollar sign and the name of the variable (no spaces). This usually makes the R code more readable.

```
mydata$height[4]
```

```
## [1] 72
```

```
mydata$height
```

```
## [1] 65 70 60 72 68
```

The colon operator can be used to generate a sequence of indices. Typing a:b will produce a range of integers a, $a+1$,..., b.

```
mydata[2:4,]
```

```
##   group age height  pets
## 2    G2  30     70  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
```

```
mydata$age[3:5]
```

```
## [1] 31 28 40
```

You can use logical conditions to obtain subsets of the data. A logical value of `TRUE` will return a data row and a logical of `FALSE` omits that row.

```
index_keep = (mydata$group =="G1")
group1 = mydata[index_keep, ]
group1
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
```

This operation can also be accomplished by using the `subset()` function.

```
group1 = subset(mydata, mydata$group == "G1")
group1
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 3    G1  31     60 FALSE
## 4    G1  28     72 FALSE
```

To subset by all values which are NOT equal to a condition we can use the logical operator !=.

```
group2 = subset(mydata, mydata$group != "G1")
group2
```

```
##   group age height pets
## 2    G2  30     70 TRUE
## 5    G2  40     68 TRUE
```

Rearranging Rows

Data can be sorted using the `order()` function. This function returns the ranks of the variable being sorted. Including more than one variable allows a "nested sort," where the second variable, third variable, etc., is used when there are ties in the sorting based on the previous variables.

```
order(mydata$height)
```

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

```
mydata[order(mydata$height),]
```

```
##   group age height  pets
## 3    G1  31     60 FALSE
## 1    G1  35     65  TRUE
## 5    G2  40     68  TRUE
## 2    G2  30     70  TRUE
## 4    G1  28     72 FALSE
```

The ranks are used as row indices for the data frame. Notice that the observations are sorted by the last column, `weight`. We can sort by more than one variable. Let's first sort by `group` alone and then by `group` followed by `weight` and see what we get.

```
mydata[order(mydata$group), ]
```

```
##   group age height  pets
## 1    G1  35     65  TRUE
## 3    G1  31     60 FALSE
```

```
## 4     G1   28      72 FALSE
## 2     G2   30      70  TRUE
## 5     G2   40      68  TRUE
```

```
mydata[order(mydata$group, mydata$height), ]
```

```
##   group age height  pets
## 3    G1   31      60 FALSE
## 1    G1   35      65  TRUE
## 4    G1   28      72 FALSE
## 5    G2   40      68  TRUE
## 2    G2   30      70  TRUE
```

To reorder a vector from smallest to largest we can also condiser the sort() function.

```
sort(mydata$age)
```

```
## [1] 28 30 31 35 40
```

Adding Columns

One can add a new variable (column) to a data frame by defining a new variable and assigning values to it. Below we add a `weight` variable to the data frame.

```
wghts = c(169, 161, 149, 165, 155)
wghts
```

```
## [1] 169 161 149 165 155
```

```
mydata$weight = wghts
mydata
```

```
##   group age height  pets weight
## 1    G1   35      65  TRUE    169
## 2    G2   30      70  TRUE    161
## 3    G1   31      60 FALSE    149
## 4    G1   28      72 FALSE    165
## 5    G2   40      68  TRUE    155
```

We can also add a new column using the `cbind()` function.

```
mydata = cbind(mydata, wghts)
mydata
```

```
##   group age height  pets weight wghts
## 1    G1   35      65  TRUE    169   169
## 2    G2   30      70  TRUE    161   161
## 3    G1   31      60 FALSE    149   149
## 4    G1   28      72 FALSE    165   165
```

```
## 5    G2  40     68  TRUE    155   155
```

NA values

In addition, if we have a missing value, or a blank value, we can use the object NA to indicate the lack of a value.

```
fav_color = c("Red", NA, "Purple", NA, "Red")

mydata = cbind(mydata, fav_color)
mydata
```

```
##  group age height  pets weight wghts fav_color
## 1    G1  35     65  TRUE    169   169       Red
## 2    G2  30     70  TRUE    161   161      <NA>
## 3    G1  31     60 FALSE    149   149    Purple
## 4    G1  28     72 FALSE    165   165      <NA>
## 5    G2  40     68  TRUE    155   155       Red
```

We can drop check for NA values using the `is.na()` function.

```
is.na(mydata)
```

```
##    group  age height  pets weight wghts fav_color
## [1,] FALSE FALSE  FALSE FALSE  FALSE FALSE   FALSE
## [2,] FALSE FALSE  FALSE FALSE  FALSE FALSE    TRUE
## [3,] FALSE FALSE  FALSE FALSE  FALSE FALSE   FALSE
## [4,] FALSE FALSE  FALSE FALSE  FALSE FALSE    TRUE
## [5,] FALSE FALSE  FALSE FALSE  FALSE FALSE   FALSE
```

We can remove all rows with NA values using `na.omit()`.

```
na.omit(mydata)
```

```
##  group age height  pets weight wghts fav_color
## 1    G1  35     65  TRUE    169   169       Red
## 3    G1  31     60 FALSE    149   149    Purple
## 5    G2  40     68  TRUE    155   155       Red
```

NULL

One can drop a variable (column) by setting it equal to the R value NULL.

```
mydata$wghts = NULL
mydata
```

```
##   group age height  pets weight fav_color
## 1    G1  35     65  TRUE    169       Red
## 2    G2  30     70  TRUE    161      <NA>
```

```
## 3    G1   31     60 FALSE    149    Purple
## 4    G1   28     72 FALSE    165      <NA>
## 5    G2   40     68  TRUE    155       Red
```

Be careful using these methods. Once a variable or row is dropped, it's gone.

Adding Rows

Rows can be added to a data frame using the `rbind()` (row bind) function. Because our columns have different data types, we will create a list object and then add it as a new row.

```
newobs = list("G1", 23, 62, FALSE, 160, "Blue")
newdata = rbind(mydata, newobs)
newdata
```

```
##   group age height  pets weight fav_color
## 1    G1   35     65  TRUE    169       Red
## 2    G2   30     70  TRUE    161      <NA>
## 3    G1   31     60 FALSE    149    Purple
## 4    G1   28     72 FALSE    165      <NA>
## 5    G2   40     68  TRUE    155       Red
## 6    G1   23     62 FALSE    160      Blue
```

We can also use `rbind()` to append one data frame to another. We can do this with the variables `group1` and `group2` created above still exist in your R environment.

```
group1
```

```
##   group age height  pets
## 1    G1   35     65  TRUE
## 3    G1   31     60 FALSE
## 4    G1   28     72 FALSE
```

```
group2
```

```
##   group age height pets
## 2    G2   30     70 TRUE
## 5    G2   40     68 TRUE
```

```
rbind(group1, group2)
```

```
##   group age height  pets
## 1    G1   35     65  TRUE
## 3    G1   31     60 FALSE
## 4    G1   28     72 FALSE
## 2    G2   30     70  TRUE
## 5    G2   40     68  TRUE
```

We can also remove rows from a data frame. Using negative index values has the same effect as including all the indices except the negative ones.

```
mydata[-c(2,4), ]
```

```
##   group age height  pets weight fav_color
## 1    G1  35     65  TRUE    169       Red
## 3    G1  31     60 FALSE    149    Purple
## 5    G2  40     68  TRUE    155       Red
```

Chapter 5

Functions

In R we have functions, functions are another type of object in R. We use functions in order to preform a series of tasks repeatedly, or preform these tasks in different settings. They can make our code much more efficient. We can build our own functions or we can use built in functions. Here we describe both types and their properties. Parts of this chapter where adapted by John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine [2016] and de Vries and Meys [2015].

5.1 Build Your Own Function

To define a function, a name is assigned and the keyword `function` is used to denote the start of the function and its argument list. Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class *function*. Functions can be passed as arguments to other functions. Functions can be nested, so that you can define a function inside another function.

Below is the **general template**

```
function_name = function(arg){
  # Function Body
  ....
  return(return_value)
}
```

In this template we have the following key components

- `function_name`: This is the actual name of the function. It is stored in R environment as an object with this name.
- `function`: A directive which tells R a function is being created.

- `arg`: An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- `function body`: The function body contains a collection of statements that defines what the function does.
- `return_value`: The output value of the function. If `return(return_value)` is not supplied then the return value of a function is the last expression in the function body to be evaluated. Your function can only return one object. This object can be any class of object you desire, like a vector, list or data frame, but you may only return one object.

Below is an example of converting a temperature from Fahrenheit to Celsius.

```
fahrenheit_to_celsius = function(temp_F){
  temp_C = (temp_F - 32) * 5 / 9
  return(temp_C)
}
```

In this example the function name is `fahrenheit_to_celsius`, there is only one input or argument, `temp_F`, and the output is the object `temp_C`.

Now if we would like to "call" this function we can simply put into the command console the function name and desired input.

```
# Convert 87F to Celsius
fahrenheit_to_celsius(temp_F = 87)
```

```
## [1] 30.55556
```

What would happen if we tried to call this function without supplying an input? This would result in an error.

```
# temp_F not defined.
fahrenheit_to_celsius()
```

```
Error in fahrenheit_to_celsius() :
  argument "temp_F" is missing, with no default
```

With functions we can define function arguments to have default values. These default values are used only if the user did not supply an argument value. Observe the example below.

```
# An example function
example_func = function(a = 1, b){
  c = a + b
  d = c + 1

  # returns a+b+1
  return(d)
}
```

```
# Call example function
example_func(a= 2, b= 3)
```

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

```
example_func(b = 3)
```

```
## [1] 5
```

Further notice that R has three ways to match function inputs to the formal arguments of the function definition. R attempts to match inputs to arguements in the following order.

1) by complete name
2) by partial name (matching on initial *n* characters of the argument name)
3) by position

After running a function command R first attempts to match arguments by complete name, then by partial name, and then by position. If it unable to match inputs to an argument it then uses the default argument value, if one exists.

Observe:

```
example_func(2, 3)
```

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

```
example_func(b = 3, a = 2)
```

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

```
example_func(a = 2, b = 3)
```

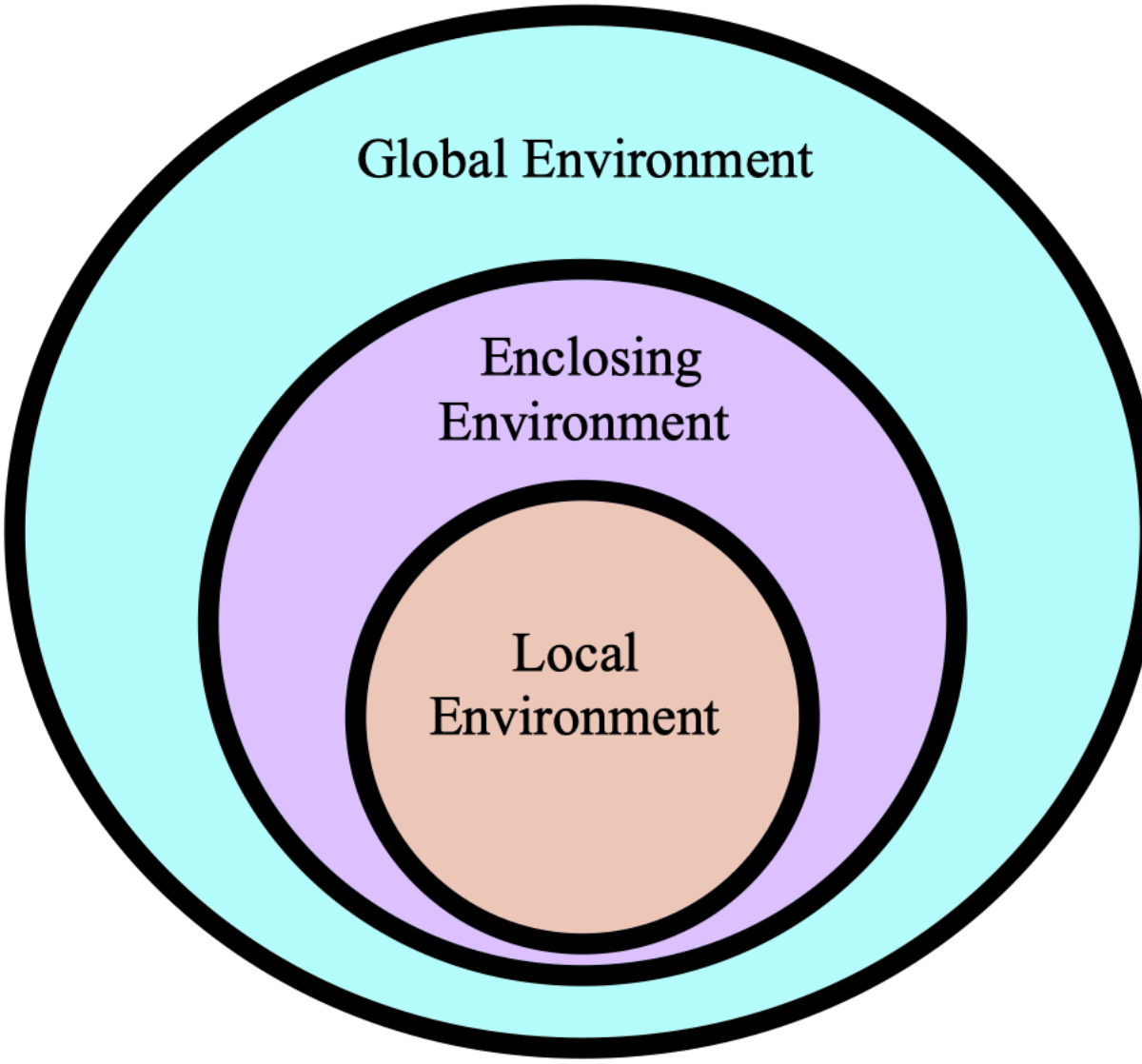
```
## [1] 6
```

5.2 Lexical Scoping

With all these examples of functions, notice that in your global environment, only the function name was added. The function arguments, return values, and all objects defined inside the function are not a part of the global environment. This is not a mistake. We can define objects locally, or temporarily, when using functions. These objects are created and used only when the function is running, and quickly discarded once the function finishes. They never are listed in the global environment.

Sometimes you may have an object defined in multiple places. When this happens, R uses a system of rules to determine which definition it will use. In

other words, how and where we define an object determines the objects *scope*, or range of places that we can use this object. The system of rules R uses for searching for objects is called *lexical scoping*, as opposed to other types of scoping. With this system R looks for objects that are called in a function within the itself, then any enclosing environments, then the global environment, and lastly looking at objects in packages or built-in objects. This is the same searching method that we see in Python and Java.


Global Environment
Enclosing Environment
Local Environment

5.3 Built-In Functions

R has functions built-in to it just like excel. You can call these built-in function at any time. We have already seen a few of these functions.

- `c()`
- `class()`
- `matrix()`
- `data.frame()`

Below are a few more examples using the built-in dataset `mtcars`, we will use `mtcars$mpg` as a vector of data to analyze.

- `mean()`: Takes in a vector, and returns the mean of the values in the vector.

```
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

- `median()`: Takes in a vector, and returns the median of the values in the vector.

```
median(mtcars$mpg)
```

```
## [1] 19.2
```

- `var()`: Takes in a vector, and returns the variance of the values in the vector.

```
var(mtcars$mpg)
```

```
## [1] 36.3241
```

- `sqrt()`: If you give it a vector, it returns the square root of each element in the vector. If you give it a single number, it returns the square root of the number.

```
sqrt(mtcars$mpg)
```

```
##  [1] 4.582576 4.582576 4.774935 4.626013 4.324350 4.254409 3.7
##  [9] 4.774935 4.381780 4.219005 4.049691 4.159327 3.898718 3.2
## [17] 3.834058 5.692100 5.513620 5.822371 4.636809 3.937004 3.8
## [25] 4.381780 5.224940 5.099020 5.513620 3.974921 4.438468 3.8
```

- `sd()`: Takes in a vector, and returns the standard deviation of the values in the vector.

```
sd(mtcars$mpg)
```

```
## [1] 6.026948
```

- `range()`: Takes in a vector, and returns the minimum AND maximum of the values in the vector.

```
range(mtcars$mpg)
```

```
## [1] 10.4 33.9
```

- `quantile()`: Takes in a vector as the first argument, and a vector of values between 0 and 1 (any number of values) for the second argument. It will return the corresponding quantiles of the values in the first vector specified by the second vector.

To get the 10^{th} and 90^{th} percentiles:

```
quantile(mtcars$mpg,c(0.10,0.90))
```

```
##   10%   90%
## 14.34 30.09
```

- `summary()`: You can give this a dataset OR a vector. It returns some summary information about the values in the dataset or vector.

```
summary(cars$speed)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     4.0    12.0    15.0    15.4    19.0    25.0
```

One of the great advantages of using R is that there is a ton of resources available to learn about it. However, this can also be a disadvantage because of the vast amount of information available. The best and first resource you should look at when trying learn more about R functions is the **Help files**.

5.4 Help Files

The Help files are in R and can be viewed from the lower right window by clicking the *Help* tab. Here you can search by function name to read about it. Each built in function has a help files, sometimes similar functions are grouped together in the same file. The R Help Files are typically the best resource to get help.

The R Help files follow a fairly standard outline. You find most of the following sections in every R Help file:

- **Title**: A one-sentence overview of the function.
- **Description**: An introduction to the high-level objectives of the function, typically about one paragraph long.
- **Usage**: A description of the syntax of the function (in other words, how the function is called). This is where you find all the arguments that you can supply to the function, as well as any default values of these arguments.

- **Arguments**: A description of each argument. Usually this includes a specification of the class (for example, character, numeric, list, and so on). This section is an important one to understand, because arguments are frequently a cause of errors in R.
- **Details**: Extended details about how the function works, provides longer descriptions of the various ways to call the function (if applicable), and a longer discussion of the arguments.
- **Value**: A description of the class of the value returned by the function.
- **See also**: Links to other relevant functions. In most of the R editors, you can click these links to read the Help files for these functions.
- **Examples**: Worked examples of real R code that you can paste into your console and run.

An alternative way to view a functions help file is by typing `?` followed by the function name, or by typing `help(function_name)`.

```
# Find a help file for the function `rep`
?rep
help(rep)
```

If you are not sure exactly which function you want, you can use `??` followed by what you believe the function name is to look at a list of functions.

```
??rep
```

5.5 The … Argument

There is a special argument function `...` that can be used. Sometimes we call a function within a function that has a different set of arguments. We can use the `...` argument as a general placeholder for passing along any additional arguments in a function call.

```
# Using the ... argument
# This function calls fun2
fun1 = function(x, ...){
  y = fun2(...)
  z = x + y
  return(z)
}

# Define fun2
fun2 = function(a){
  b = a^2
  return(b)
```

```
}

# Returns 1 + 2^2 = 5
fun1(1, 2)
```

```
## [1] 5
```

We can also use the `...` arguement in a nested fashion.

```
fun1 = function(x, ...){
  y = fun2(...)
  z = x+ y
  return(z)
}

fun2 = function(a, ... ){
  b = fun3(...)
  c = a^2 + b
  return(c)
}

fun3 = function(n){
  n = sqrt(n)
  return(n)
}

#  Returns:  1 + 2^2 + sqrt(4) = 7
fun1(1, 2, 4)
```

```
## [1] 7
```

```
# Returns:  1 + 2^2 + sqrt(4) = 7
# Define each input explicitly
fun1(x = 1, a = 2, n = 4)
```

```
## [1] 7
```

When using the `...` argument it is customary to always place `...` at the end of a function argument list, as it can be difficult for R to determine which arguments are to be passed to the next function. If possible, it is good practice to explicitly define the arguments to avoid an error.

5.6 Additional Resources

- Chapter 14 of “R for Programming in Data Science”
- Videos:
  - R Functions 1 | Anatomy of a Function

- R Functions 2 | Getting Help
- R Functions 3 | Argument Order and Default Values
- Writing Functions 1 | Introduction
- Writing Functions 2 | 4 Fundamental Steps

Chapter 6

If Statements

If statements are a type of control flow structure. Control structures help us control how many times code is executed, and when it will be executed. This is helpful because sometime we only want our program to run if certain conditions are met.

6.1 If

In R we can also execute commands only if certain conditions are met using the `if` operator. This feature in R checks a logical value (`<value>`) and if it is `TRUE` then a sequence of commands within `{}` will be ran. If `<value>` is `FALSE`, then the commands inside of `{}` will not be ran.

```
if(<value>){
  # Commands
}
```

Below we have an example. The code will only be executed if `x` is a positive number.

```
x = 3

if(x>0){
  type = "positive"
}
type
```

```
## [1] "positive"
```

In the example above we have a relational operator which returns a logical value. This logical value was equal to `TRUE` so the code was executed.

6.2 If Else

We can pair an `if` statement with an `else` value. After the `else` object we can define another sequence of commands inside of `{}`. The `else` value is paired with the immediate previous `if` statement. If this `if` statement is `FALSE` then the `else` code will run. If the `if` statement is `TRUE` then the `else` code will not be executed.

```
x = -3

if(class(x) == "numeric"){
  type = "number"
} else {
  type = "not a number"
}
type
```

```
## [1] "number"
```

6.3 Else If

Sometimes we will want to do a sequence of checks that are all related, and we will only want code to run if the previous `if` statements were `FALSE` and another criteria is `TRUE`. We can use `else if` to implement these rules.

```
x = -3


if(x>0){
    type = "positive"
} else if (x <0) {
    type = "negative"
} else if(x == 0){
    type = "zero"
} else {
    type = "Error"
}

type
```

```
## [1] "negative"
```

The command for the `if` statement will only run if `x>0`, and the rest of the code will not be implemented.

The first `else if` commands will only execute if the first `if` statement was `FALSE` and `x<0`.

The second `else if` commands will only execute if the previous `else if` and `if` statements were `FALSE` and `x==0`.

If the previous `if` statement and all previous `else if` statements are `FALSE` then the `else` code will be executed.

Here is another example with an `if else` chain.

```
Toyfun = function(X,Y,Do){
  if(Do == "Add"){
    Z = X+Y

  }else if(Do =="Subtract"){
    Z = X-Y

  }else if(Do =="Multiply"){
    Z = X*Y


  }else if(Do =="Penguin"){
    Z = c("<('' )")

  } else{
    Z = c(X,Y)
  }

  return(Z)
}
Toyfun(2,4,"Add")
```

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

```
Toyfun(2,4,"Subtract")
```

```
## [1] -2
```

```
Toyfun(2,4,"Penguin")
```

```
## [1] "<('' )"
```

```
Toyfun(2,4, "Typo")
```

```
## [1] 2 4
```

6.4 Ifelse

We have a shortcut function that can help us keep our code more succinct using the `ifelse()` function. The first argument of this function is a logical value.

the second and third arguments are what to do if the value is `TRUE` or `FALSE`, respectively.

```
x = 3
type = ifelse(x>0,  "positive", "nonpositive")
type
```

```
## [1] "positive"
```

We can also pass `ifelse()` a vector, and it will check the logical condition for each element of the vector. For example, in the `mtcars` data set, we can find the proportion of cars have `mpg > 25` and `hp > 60`

```
fast_efficient = ifelse(mtcars$mpg > 25 & mtcars$hp>60, TRUE,
sum(fast_efficient)/length(fast_efficient)
```

```
## [1] 0.15625
```

6.5 Nested If Chains

We can make `if-else` chains nested within each other.

```
x = 105
if(x>0){
  if(x>100){
     type = "large positive number"
  } else {
     type = "positive number"
  }
} else if(x<0) {
  type = "negative number"
} else if(x==0){
  type = "zero"
}else {
  type = "Error"
}

type
```

```
## [1] "large positive number"
```

6.6 Additional Resources

- Chapter 13 of “R for Programming in Data Science”
- Videos:
  - Control Flow 1 | if Statements

- Control Flow 2 | if else Statements
- Control Flow 3 | else if Statements
- Control Flow 4 | ifelse Function
- Control Flow 5 | switch Function

Chapter 7

Base R Plotting

7.1 Load A Big Data Set

Let use some of the methods above, and others, to analyze a real data set. The Behavioral Risk Factor Surveillance System (BRFSS) is an annual telephone survey of 350,000 people in the United States. As its name implies, the BRFSS is designed to identify risk factors in the adult population and report emerging health trends. For example, respondents are asked about their diet and weekly physical activity, their HIV/AIDS status, possible tobacco use, and even their level of healthcare coverage. The BRFSS Web site (http://www.cdc.gov/brfss) contains a complete description of the survey, including the research questions that motivate the study and many interesting results derived from the data. We will focus on a random sample of 20,000 people from the BRFSS survey conducted in the year 2000. While there are over 200 variables in this data set, we will work with a smaller subset.

We begin by loading the data set of 20,000 observations into the R workspace and examine some of its attributes.

```
source("http://www.openintro.org/stat/data/cdc.R")
```

After a brief time, a new data frame cdc appears in the workspace. Each row representing a case (a person surveyed) and each column representing a variable.

To get general information on each variable, use the `summary()` function.

```
summary(cdc)
```

```
##     genhlth      exerany       hlthplan       smoke100
##  excellent:4657  Min.   :0.0000  Min.   :0.0000  Min.   :0.0000
##  very good:6972  1st Qu.:0.0000  1st Qu.:1.0000  1st Qu.:0.0000
```

```
##  good    :5675  Median :1.0000  Median :1.0000  Median :0.000
##  fair    :2019  Mean   :0.7457  Mean   :0.8738  Mean   :0.4721
##  poor    : 677  3rd Qu.:1.0000  3rd Qu.:1.0000  3rd Qu.:1.000
##                 Max.   :1.0000  Max.   :1.0000  Max.   :1.0000
##     height          weight        wtdesire          age          gender
##  Min.   :48.00   Min.   : 68.0   Min.   : 68.0   Min.   :18.00   m:
##  1st Qu.:64.00   1st Qu.:140.0   1st Qu.:130.0   1st Qu.:31.00
##  Median :67.00   Median :165.0   Median :150.0   Median :43.00
##  Mean   :67.18   Mean   :169.7   Mean   :155.1   Mean   :45.07
##  3rd Qu.:70.00   3rd Qu.:190.0   3rd Qu.:175.0   3rd Qu.:57.00
##  Max.   :93.00   Max.   :500.0   Max.   :680.0   Max.   :99.00
```

The variables `genhlth` and `gender` are character variables. The `summary()` command reports the frequencies of the unique values. The variables `exerany`, `hlthplan`, and `smoke100` are yes/no variables coded as 1=yes or 0=no. They represent the existence or absence or regular exercise, the presence of a healthcare plan, and whether or not the person smoked 100 cigarettes in their lifetime. The means are the proportion of "yes" responses. The variables `height`, `weight`, `wtdesire`, and `age` are numeric variables. The `summary()` command gives information on the means, medians, quartiles and range of values.

Since this is a very large data set, we wouldn't want to list all the data. We can use the functions `head()` and `tail()` to list the first and last few rows.

```
head(cdc)
```

```
##    genhlth exerany hlthplan smoke100 height weight wtdesire ag
## 1     good       0        1       0     70    175      175  77     m
## 2     good       0        1       1     64    125      115  33     f
## 3     good       1        1       1     60    105      105  49     f
## 4     good       1        1       0     66    132      124  42     f
## 5 very good      0        1       0     61    150      130  55     f
## 6 very good      1        1       0     64    114      114  55     f
```

```
tail(cdc)
```

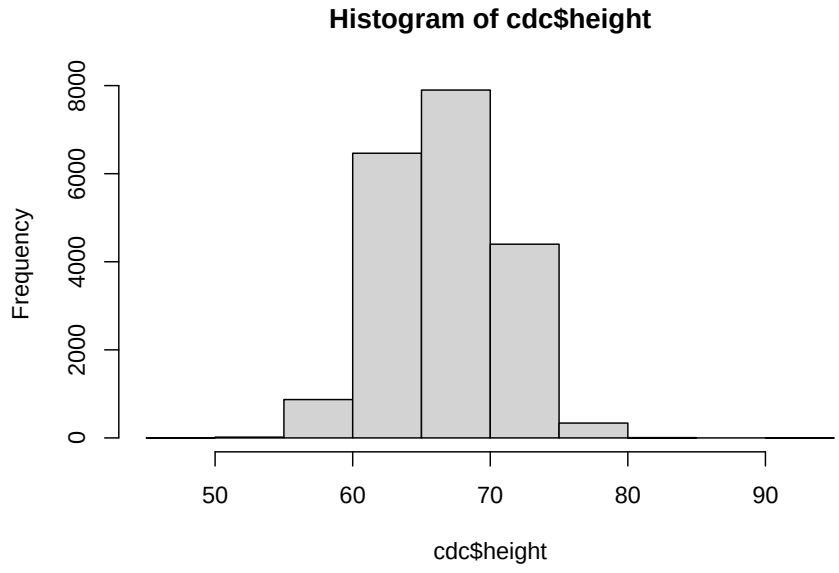
```
##       genhlth exerany hlthplan smoke100 height weight wtdesire
## 19995    good       0        1       1     69    224      224  73     m
## 19996    good       1        1       0     66    215      140  23     f
## 19997 excellent      0        1       0     73    200      185  35     m
## 19998    poor       0        1       0     65    216      150  57     f
## 19999    good       1        1       0     67    165      165  81     f
## 20000    good       1        1       1     69    170      165  83     m
```

7.2 Histograms

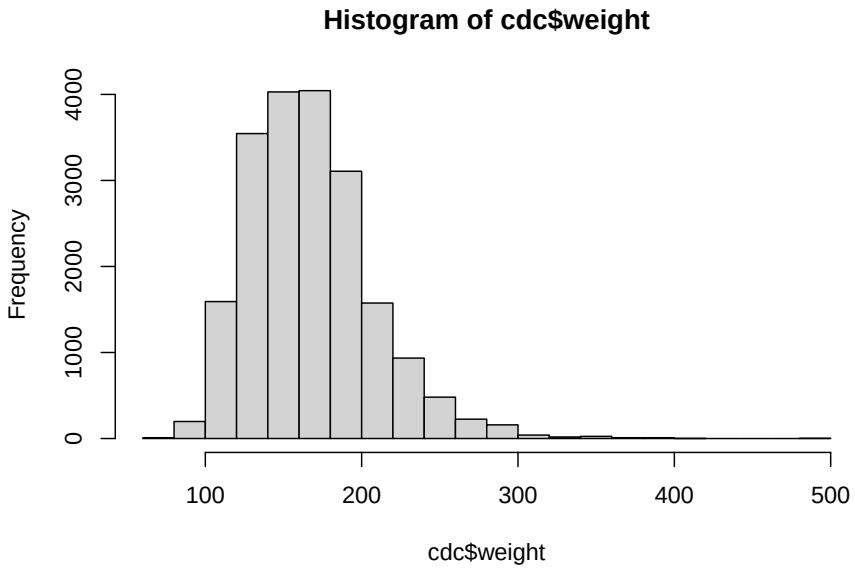
Histograms are one of the fundamental ways that we can represent a data set. In a histogram we typically have the frequency or proportion on the y-axis, and the x-axis is segmented into mutually exclusive sections. The height of the bins corresponds to amount of observations that fall within a specific range. We can create a histogram using the `hist()` function.

Most of the basic R plots have default settings. For example, the x-axis, y-axis, number of bins, titles, and labels all change depending on the vector supplied into the function.

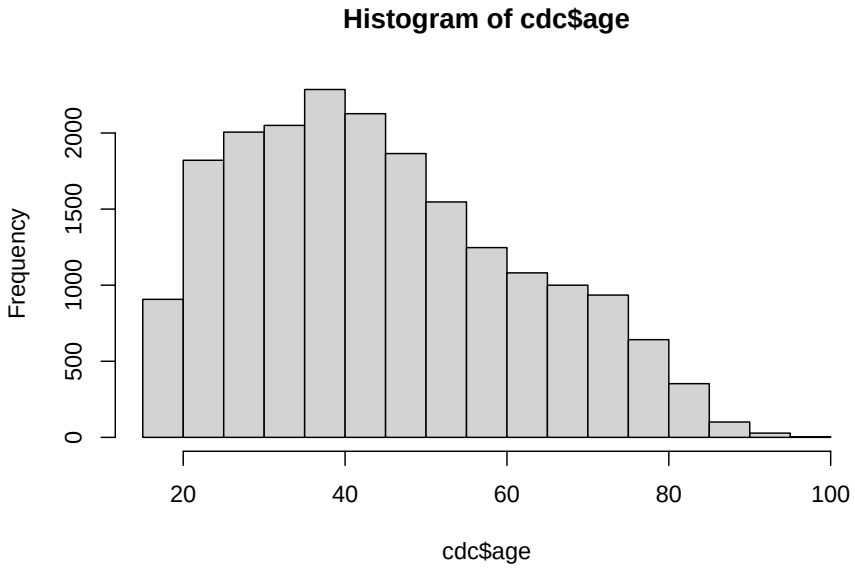
```
# Look at the help file
?hist


hist(cdc$height)
```




```
hist(cdc$weight)
```



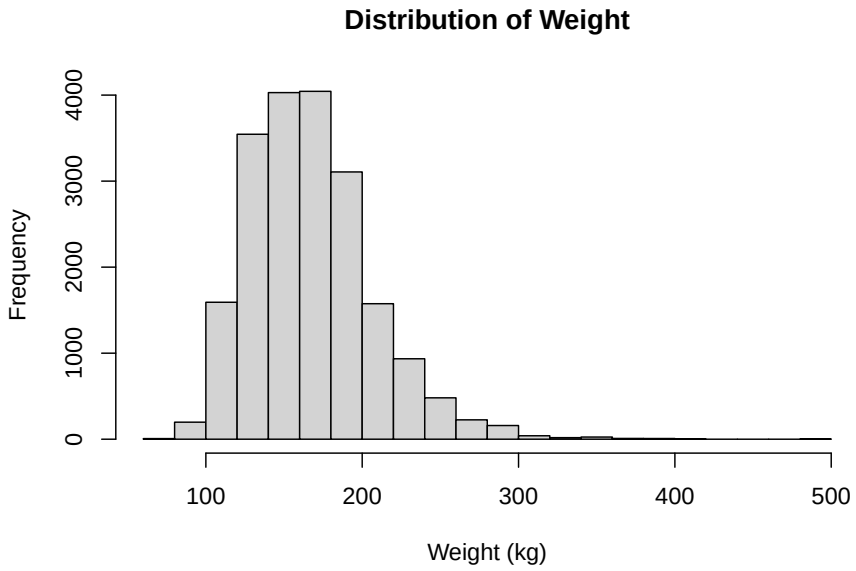

```
hist(cdc$age)
```



The output appears in the *Plots* panel of *RStudio*. You can use the arrows to the left of the *Zoom* button the switch among the three plots.

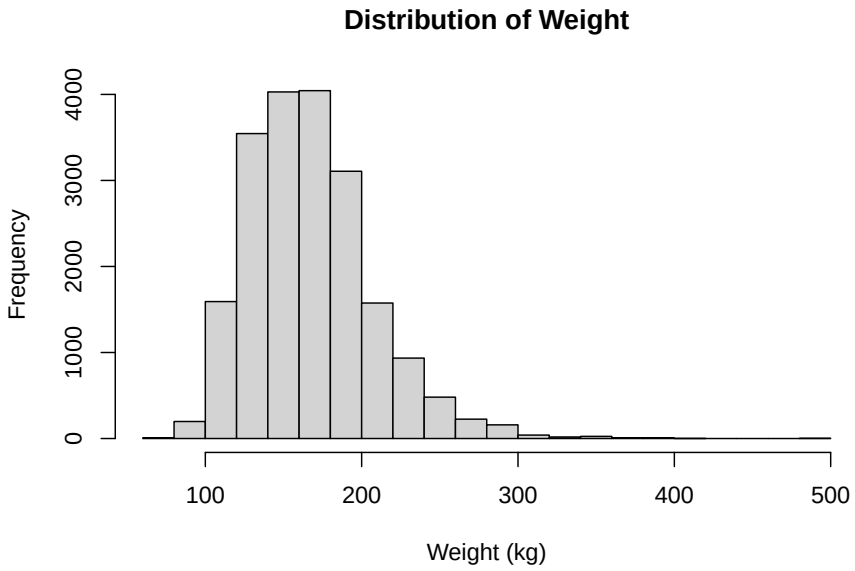
There are several settings in base R plots that are similar. For example, in base R plots typically we can change the title, x-axis label, and y-axis label with `main`, `xlab`, and `ylab` arguments.

```
hist(cdc$weight, main="Distribution of Weight",xlab="Weight (kg)")
```



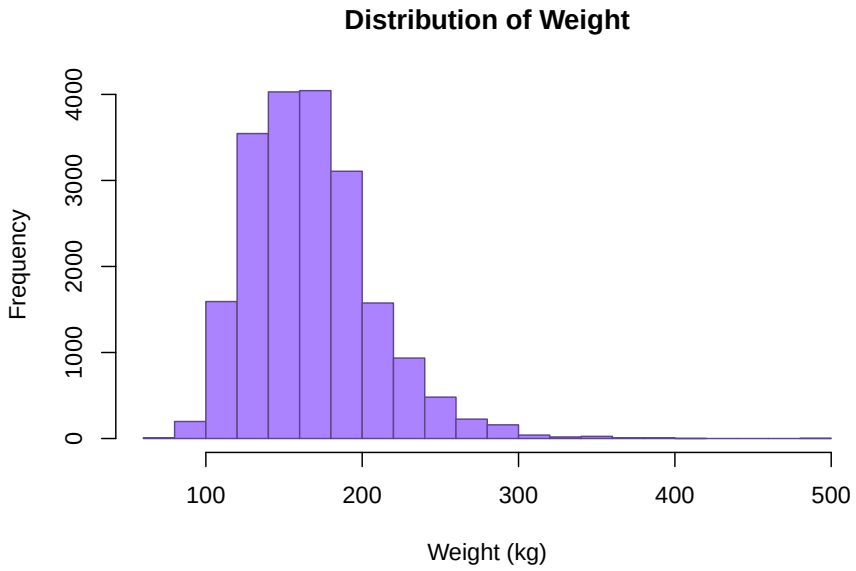
There are also function specific arguments. For example, we can control the number of bins to create.

```
hist(cdc$weight, main="Distribution of Weight",xlab="Weight (kg)",
     breaks = 20)
```



Use `col` argument to change the colors used for the bars. By using the `border` argument, you can even change the color used for the border of the bars.

```
hist(cdc$weight, breaks=20, main="Distribution of Weight",
     xlab="Weight (kg)",
     border = "mediumpurple4",
     col = "mediumpurple1")
```



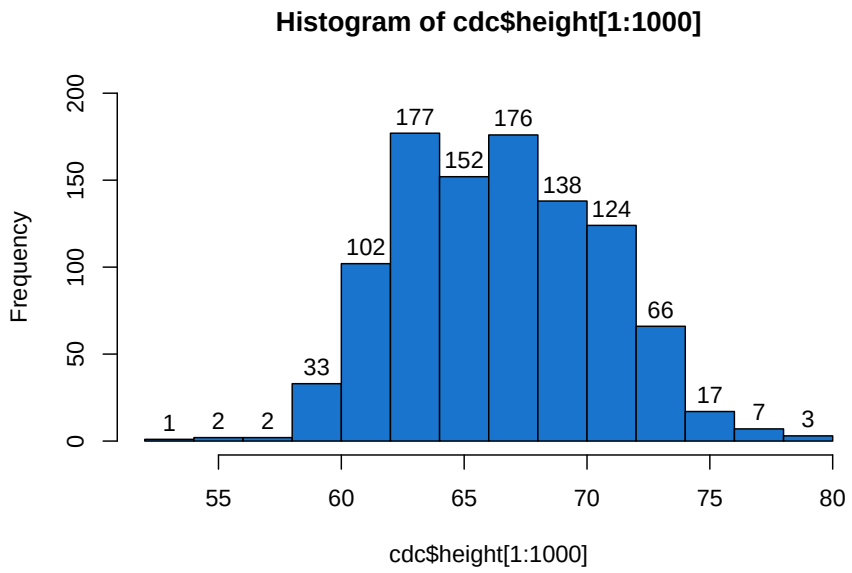
There are several ways we can add **colors** to R.

- **Using Color Names**: R programming has names for 657 colors. You can take a look at them all with the `colors()` function, or simply check this R color pdf.
- **Using Hex Values as Colors**: Instead of using a color name, color can also be defined with a hexadecimal value. We define a color as a 6 hexadecimal digit number of the form `#RRGGBB`. Where the `RR` is for red, `GG` for green and `BB` for blue and value ranges from `00` to `FF`. For example, `#FF0000` would be red and `#00FF00` would be green similarly, #FFFFFF would be white and `#000000` would be black.
- **Using RGB Values** The function `rgb()` allows us to specify red, green and blue component with a number between 0 and 1. This function returns the corresponding hex code discussed above.
- **Using a Color Palette**: R programming offers 5 built in color palettes which can be used to quickly generate color vectors of desired length. They are: `rainbow()`, `heat.colors()`, `terrain.colors()`,`topo.colors()` and `cm.colors()`. We pass in the number of colors that we want

You can also place values on top of bars; which will help you interpret the graph correctly. You can add them by setting the labels argument to `TRUE`.

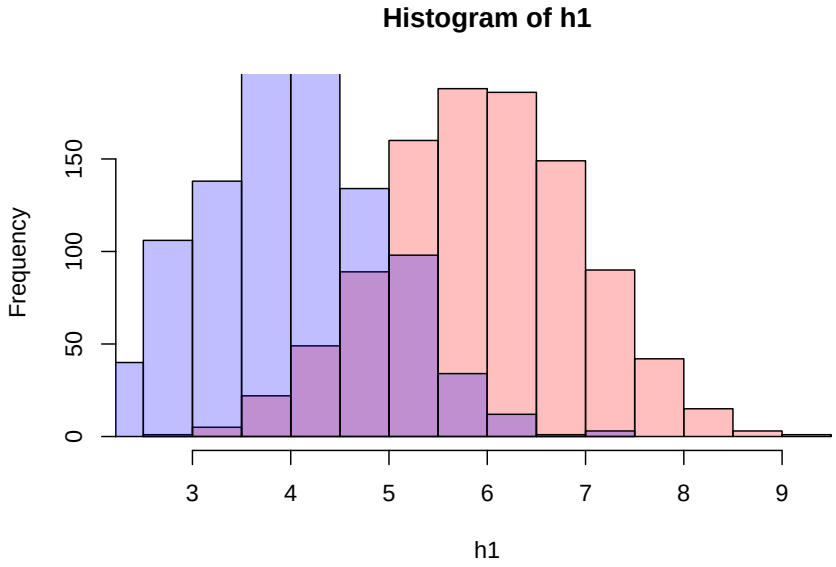
```
hist(cdc$height[1:1000],
     col="dodgerblue3",
```

```
      labels=TRUE,
      ylim = c(0, 200),
      breaks = 18)
```



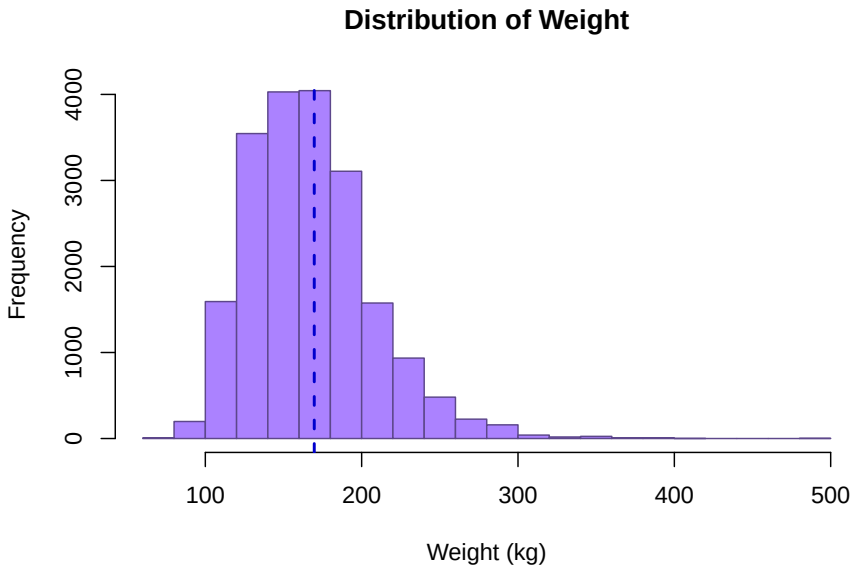
Often you want to compare the distributions of different variables within your data. You can overlay the histograms by setting the `add` argument of the second histogram to `TRUE`.

```
# random numbers
h1 = rnorm(1000,6)
h2 = rnorm(1000,4)

# Overlay two histograms
hist(h1,
      col=rgb(1,0,0,0.25))
hist(h2,
      col=rgb(0,0,1,0.25),
      add=TRUE)
```



Often you want to draw attention to specific values or observations in your graphic to provide unique insight. You can do this by adding markers to your graphic. For example, adding mean line will give you an idea about how much of the distribution is above and below the average. You can add such marker by using the `abline()` function.

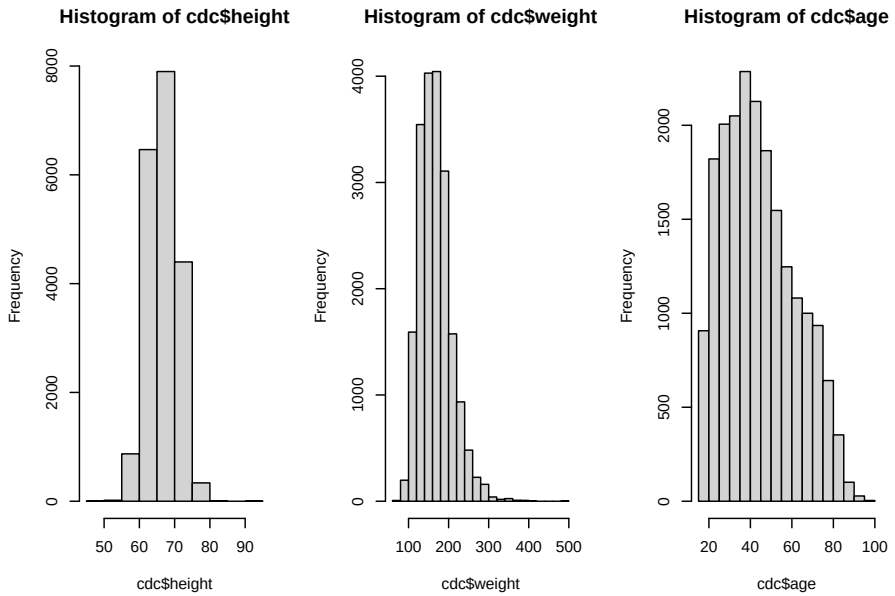
```
hist(cdc$weight, breaks=20, main="Distribution of Weight",
     xlab="Weight (kg)",
     border = "mediumpurple4",
     col = "mediumpurple1")

abline(v=mean(cdc$weight),
       col="mediumblue",
       lty=2,
       lwd=2)
```



Notice that the `abline()` function had arguments `lty`, and `lwd`. The description for these arguments is in the `par` help file which contains a master set of graphical parameter arguments. In this file we can also see that `main`, `xlab`, and `ylab` are also listed.

The `par` file contains graphical arguments that are common in base R graphics functions, and it is a function that can adjust global plotting window parameters. For example, suppose we wish to have multiple plots on plotting window. We can control this with `mfrow`. When adjusting global plotting parameters it is best to always revert back to the default settings when you are done.

```
# Change plot window to have 3 columns and 1 row of plots
par(mfrow = c(1, 3))

# Three plots in one window
hist(cdc$height)
hist(cdc$weight)
hist(cdc$age)
```



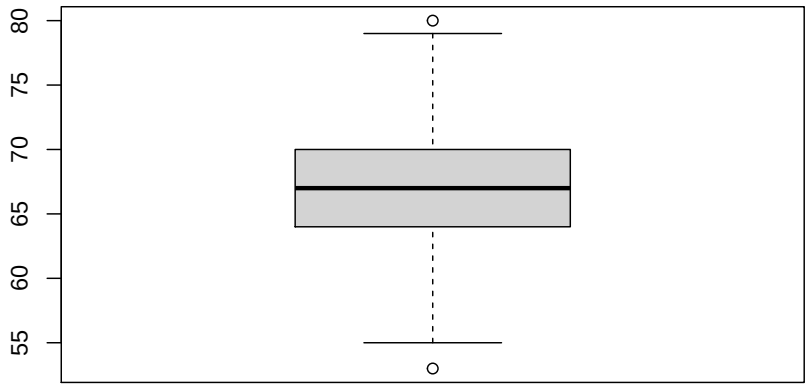

```
# Revert back to default
par(mfrow = c(1,1))
```

For more options, look up “hist” in the *Help* panel of *RStudio*.

7.3 Boxplot

Let’s produce a boxplot for the first 1000 values of the height variable.

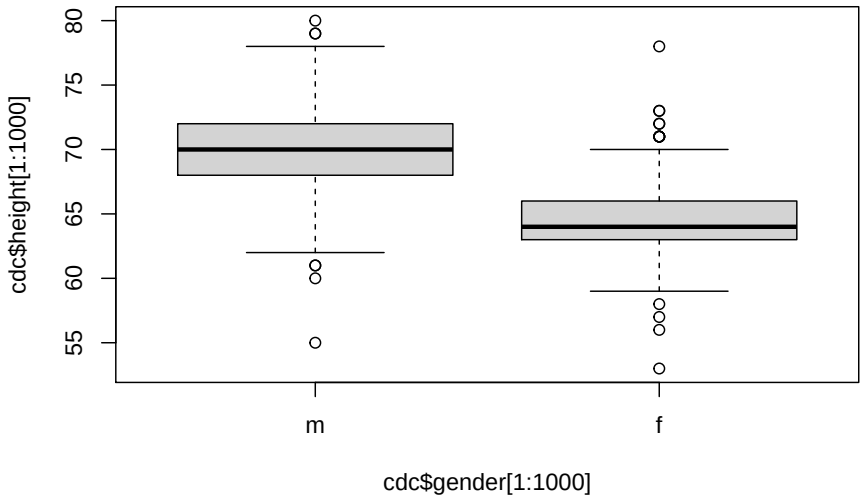
```
boxplot(cdc$height[1:1000])
```



The line in the center is the median. The bottom and top of the box are drawn at the first (Q_1) and third (Q_3) quartiles (same as the 25th and 75th percentiles). The difference between the third and first quartiles is called the interquartile range ($Q_3 - Q_1$). This is the height of the box. The lines above and below the box are called the whiskers. The upper whisker is either the third quartile plus 1.5 times the interquartile range, $Q3 + 1.5(Q_3 - Q_1)$, or the largest data value, whichever is smallest. Similarly, the lower whisker is either the first quartile minus 1.5 times the interquartile range, $Q1 - 1.5(Q_3 - Q_1)$, or the smallest data value, whichever is largest. If data values exceed the whiskers, they are plotted as circles. Boxplots are often used to represent numeric data.

One can use boxplots to compare different groups using ~ character. On the right side of ~ is the numeric variable, and the left side of ~ is a grouping variable (character, logical, factor).

```
boxplot(cdc$height[1:1000] ~ cdc$gender[1:1000])
```



7.4 Scatter Plot

7.5 Pie Charts

We will now look at some of the qualitative data that are not numbers, but categories or groups. The `table()` function can be used to tabulate categorical data. The `genhlth` variable has five categories, we can use `table()` to find the frequencies.

```
table(cdc$genhlth)
```

```
##
## excellent very good      good      fair      poor
##      4657      6972      5675      2019       677
```

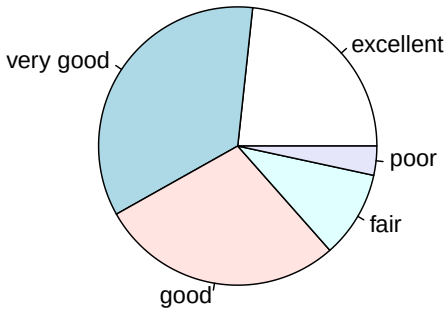
Since the sample size is 20,000, we can divide by n to get proportions.

```
table(cdc$genhlth)/20000
```

```
##
## excellent very good      good      fair      poor
##   0.23285   0.34860   0.28375   0.10095   0.03385
```

Pie charts are also used for categorical data. Options are also available for the `pie()` function.

```
pie(table(cdc$genhlth)/20000)
```



Options are also available for the pie() function.

```
colors = c("green", "blue", "yellow", "pink", "red")
pie(table(cdc$genhlth)/20000, col = colors, main = "General He
```


General Health
very good
excellent
poor
fair
good

Chapter 8

Packages

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at R Packages.

To see a list of all packages installed on your device.

```
library()
```

To see a list of all packages that are currently loaded (note that yours will likely look different).

```
search()
```

```
##  [1] ".GlobalEnv"      "package:knitr"   "package:stats"
##  [4] "package:graphics" "package:grDevices" "package:utils"
##  [7] "package:datasets" "package:methods"  "Autoloads"
## [10] "package:base"
```

When adding a new package to our library we only have to install it once. We can do so with the following command.

```
install.packages("library name")
```

Alternatively, we can also go to the lower left hand window and select the *Packages* tab. Then hit the button **Install**. A dropdown menu will appear where we can search for the package name.

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

```
library("library name")
```

For example, suppose we wanted to install and load the package "ggplot2", (a very popular package for making plots). We would type the following commands.

```
# Install package (only need to do this once)
install.packages("ggplot2")

# Load into working environment (need to do this for each new
library("ggplot2")
```

It might seem strange to (1) have to download packages tto use features and (2) have to load these packages each time we wish to use them; however, there are several good reasons for doing packages in this way, and this system is considered a feature. We don't have all possible packages available to us at all times because that is a lot of information that R would need to store at once which could make our computer lag. These packages are also always being updated at different rates with different features. Most users will only utilize packages in CRAN, so having to update R each session for features you may never use would be tedious. Furthermore, there are several packages which contain objects and functions with the same name.

8.1 Namespace Collisions

Every time we load a package into our environment the results for the `search()` function changes. The most recently added package is always listed after the global environment, followed by the second most recent, and so on. The `search()` function tells us how R searches for an object that we called. For example, consider the following command.

```
v = c(1,2,3)
mean(v)
```

```
## [1] 2
```

In this case R is searching for the function mean first in the global environment, and then in the package in the order that appears in the output of the `search()` function. The package `base` is always last. In our case the function `mean()` is only defined in the base package. If you want to call a function from a specific package explicitly and want to be sure there is no confusion you can type the package name followed by `::` and the desired command.

```
base::mean(v)
```

```
## [1] 2
```

This can be useful, but is not usually necessary.

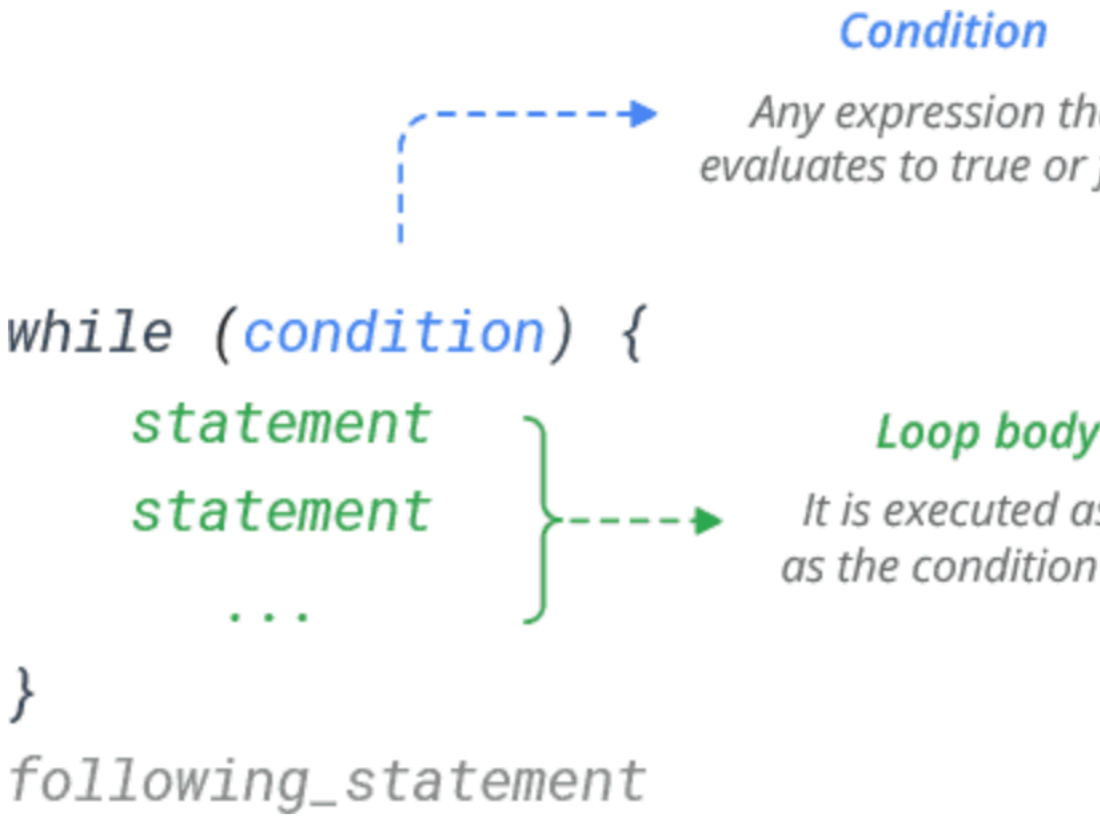
Chapter 9

Loops

Loops are another type of control flow structure. They dictate how many times a series of commands should be run. There are two type of loops, a `while` loop and a `for` loop. These two loops operate similarly and are found in other programming languages such as `C` and `Python`.

9.1 While Loop

A `while` loop is used when you want to perform a task indefinitely, until a particular condition is met. It's a condition-controlled loop.



The loop will continue until the condition is `FALSE`.

```
x = 5

# If statement is true, keep running the loop
while (x != 0 ) {
  print(x)
  x = x - 1
}
```

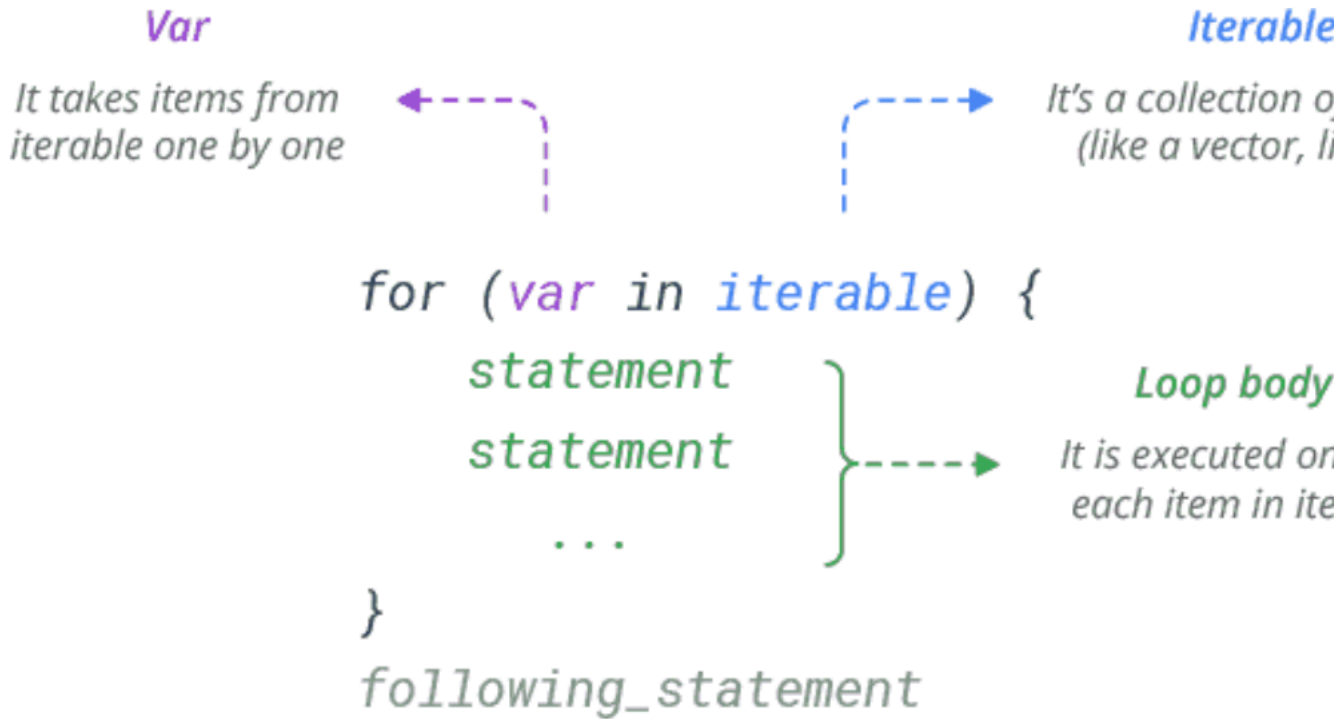
```
## [1] 5
## [1] 4
## [1] 3
## [1] 2
## [1] 1
```

If the condition is false at the start, the while loop will never be executed at all.

```
x = 0

# If statement starts as TRUE,  the loop will never run
while (x != 0 ) {
  print(x)
  x = x - 1
}
```

9.2 For Loops

The `for` statement in R is a bit different from what you usually use in other programming languages. Rather than iterating over a numeric progression, R's `for` statement iterates over the items of a vector or a list. The items are iterated in the order that they appear in the vector.




```
# Iterate through a vector
colors = c("red","green","blue","yellow")
```

```
for (x in colors) {
  print(x)
}
```

```
## [1] "red"
## [1] "green"
## [1] "blue"
## [1] "yellow"
```

```
lst = list(3.14, "Hi", c(1,2,3))

for (i in lst) {
  print(i)
}
```

```
## [1] 3.14
## [1] "Hi"
## [1] 1 2 3
```

If you need to execute a group of statements for a specified number of times, use sequence operator : or built-in function `seq()`.

```
# Print 'Hello!' 3 times
for (x in 1:3) {
  print("Hello!")
}
```

```
## [1] "Hello!"
## [1] "Hello!"
## [1] "Hello!"
```

```
for (x in seq(from=2,to=8,by=2)) {
  print(x^2)
}
```

```
## [1] 4
## [1] 16
## [1] 36
## [1] 64
```

9.3 Break

The `break` statement is used to exit the loop immediately. It simply jumps out of the loop altogether, and the program continues after the loop.

```
x = 5
```

```
# If statement starts as TRUE,  the loop will never run
while (x != 0 ) {
  print(x)
  x = x - 1

  if(x == 2){
    print("Entered IF statement, stop loop")
    break
  }

}
```

```
## [1] 5
## [1] 4
## [1] 3
## [1] "Entered IF statement, stop loop"
```

If not given an adequate stopping criteria or break statement the loop will continue forever. For example, if we started the above examples at `x = -2`. The `break` statement is particularly important for the `while` loop.

The `break` statement can also be used in a `for` loop. It simply jumps out of the loop altogether, and the program continues after the loop.

```
colors = c("red","green","blue","yellow")
for (x in colors) {
  if (x == "blue"){
       break
  }
  print(x)
}
```

```
## [1] "red"
## [1] "green"
```

The `for` loops do not have the same risk of "running forever", like `while` loops have.

9.4 Next

We can also use the `next` command in both `for` loops and `while` loops in order to skip executing a command.

```
for(i in 1:10){

  i = i^2
```

```
  if(i <5){
    next
  }

  print(i)
}
```

```
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

The code inside the loop will run until it reaches the `next` statement. Once

9.5 Bisection

9.6 Nested Loops

9.7 Additional Resources

- Chapter 13 of “R for Programming in Data Science”
- Videos:
  - Control Flow 6 | while loops
  - Control Flow 7 | while loops Bisection Method
  - Control Flow 8 | infinite while loops
  - Control Flow 9 | for loop Intro
  - Control Flow 10 | break and next
  - Control Flow 11 | nested for loops
  - Programming Loops vs. Recursion - Computerphile

Chapter 10

Apply Family of Functions

Loops (like `for`, and `while`) are a way to repeatedly execute some code. However, they are often slow in execution when it comes to processing large data sets.

`R` has a more efficient and quick approach to perform iterations – **The apply family**.

The apply family consists of vectorized functions. Below are the most common forms of apply functions.

- `apply()`
- `lapply()`
- `sapply()`
- `tapply()`

These functions let you take data in batches and process the whole batch at once.

There primary difference is in the object (such as list, matrix, data frame etc.) on which the function is applied to and the object that will be returned from the function.

10.1 apply() function

The `apply()` function is used to apply a function to the rows or columns of matrices or data frames. It assembles the returned values into a vector, and then returns that vector.

If you want to apply a function on a data frame, make sure that the data frame is homogeneous (i.e. either all numeric values or all character strings) Otherwise,

R will force all columns to have identical types. This may not be what you want. In that case, use the `lapply()` or `sapply()` functions.

Description of the required `apply()` arguments:

- `X`: A matrix , data frame or array
- `MARGIN`: A vector giving the subscripts which the function will be applied over.
  - 1 indicates rows
  - 2 indicates columns
  - c(1, 2) indicates rows and columns
- `FUN`: The function to be applied

```
# Get column means
data = matrix(1:9, nrow=3, ncol=3)
data
```

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

```
apply(data, 2, mean)
```

```
## [1] 2 5 8
```

```
# Get row means
apply(data, 1, sum)
```

```
## [1] 12 15 18
```

You can use user-defined functions as well.

```
apply(data, 2, function(x){

  # Standard deviation formula
  y = sum(x -mean(x))^2/(length(x)-1)

  return(y)
  })
```

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

10.2 lapply() function

The `lapply()` function is used to apply a function to each element of the list. It collects the returned values into a list, and then **returns that list**.

Description of the required `lapply()` arguments:

- X: A matrix , data frame or array
- FUN: The function to be applied

```
data_lst = list(item1 = 1:5,
                item2 = seq(4,36,8),
                item3 = c(1,3,5,7,9))
data_lst
```

```
## $item1
## [1] 1 2 3 4 5
##
## $item2
## [1]  4 12 20 28 36
##
## $item3
## [1] 1 3 5 7 9
```

```
data_vector = c(1,2,3,4,5,6,7,8)
data_vector
```

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

```
lapply(data_lst, sum)
```

```
## $item1
## [1] 15
##
## $item2
## [1] 100
##
## $item3
## [1] 25
```

```
lapply(data_vector, sum)
```

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

```
## [1] 5
##
## [[6]]
## [1] 6
##
## [[7]]
## [1] 7
##
## [[8]]
## [1] 8
```

10.3 sapply() function

The `sapply()` and `lapply()` work basically the same.

The only difference is that `lapply()` always returns a list, whereas `sapply()` tries to simplify the result into a vector or matrix.

- If the return value is a list where every element is length 1, you get a vector.
- If the return value is a list where every element is a vector of the same length (> 1), you get a matrix.
- If the lengths vary, simplification is impossible and you get a list.

Description of the required `sapply()` arguments:

- `X`: A matrix , data frame or array
- `FUN`: The function to be applied

```
data_lst = list(item1 = 1:5,
                item2 = seq(4,36,8),
                item3 = c(1,3,5,7,9))
data_lst
```

```
## $item1
## [1] 1 2 3 4 5
##
## $item2
## [1]  4 12 20 28 36
##
## $item3
## [1] 1 3 5 7 9
```

```
sapply(data_lst, sum)
```

```
## item1 item2 item3
##    15   100    25
```

10.4 tapply() function

The `tapply()` function breaks the data set up into groups and applies a function to each group.

Description of the required `sapply()` arguments:

- `X`: A matrix , data frame or array
- `INDEX`: A grouping factor or a list of factors
- `FUN`: The function to be applied

```
data = data.frame(name=c("Amy","Max","Ray","Kim","Sam","Eve","Bob"),
                  age=c(24, 22, 21, 23, 20, 24, 21),
                  gender=factor(c("F","M","M","F","M","F","M")))

data
```

```
##   name age gender
## 1  Amy  24      F
## 2  Max  22      M
## 3  Ray  21      M
## 4  Kim  23      F
## 5  Sam  20      M
## 6  Eve  24      F
## 7  Bob  21      M
```

```
tapply(data$age, data$gender, min)
```

```
##  F  M
## 23 20
```

Bibliography

Dr. Robert Desharnais. *Biology3000*. California State University, Los Angeles, Los Angeles, CA, 2020.

Andrie de Vries and Joris Meys. *R for dummies*. John Wiley & Sons, Inc, Hoboken, New Jersey, 2015. URL https://www.dummies.com/programming/r/how-to-use-the-r-help-files/. ISBN 978-1119055808.

John Blischak, Daniel Chen, Harriet Dashnow, and Denis Haine. Software carpentry: Programming with r.version 2016.06, 2016. URL https://swcarpentry.github.io/r-novice-inflammation/02-func-R/index.html.