Getting StaRted with R

Ryan Carlson and Xanni Brown

2022-03-06

Contents

Figure 1: img

Chapter 1

The value of learning R

Welcome! This is a guide for learning statistics in R via R Studio. R is a powerful, open-source programming language for data analysis and visualization, and the number one programming language in psychological science and data science. R Studio is a stylish user interface for coding in R, which makes learning R intuitive.

Why learn R? Here's a few reasons:

- To efficiently analyze and visualize data for a project you're working on
- To collaborate and promote open science, allowing others to efficiently analyze and visualize your data too
- To gain a marketable programming experience
- To learn a new language and way of thinking about problems

To get staRted with R, continue to the *Setup* page!

Chapter 2

Set-up

2.1 Installing R

R is a free software, which you can download and install at:

http://cran.r-project.org/

When you open this page, you will see a box at the top labeled "Download and Install R" with three different links for Linux, Mac, and Windows users. Choose the one that's appropriate for you, and then follow the instructions on the page to install the latest version.

2.2 Installing R Studio

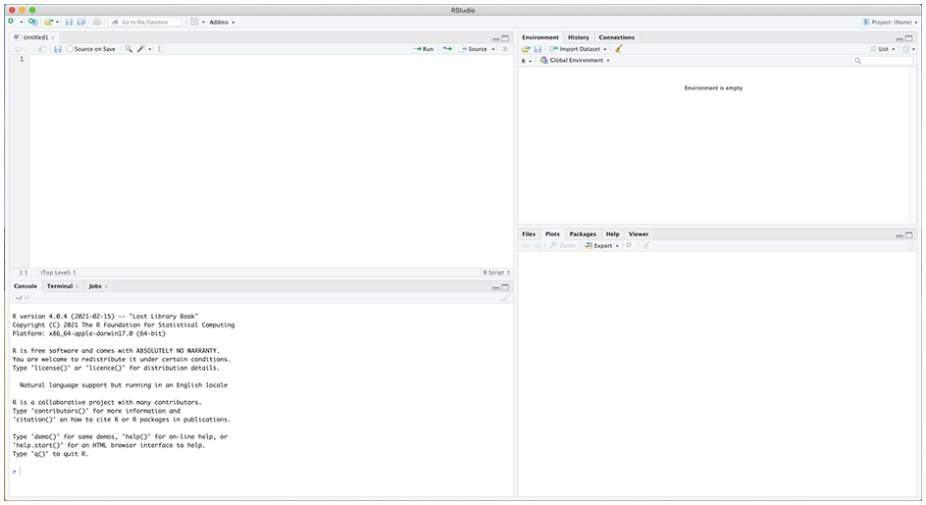
Once you have done that, the next step is to install RStudio. RStudio provides a helpful and functional interface, which will allow you to write and edit your R code much more effectively. RStudio is also a free software (though the developers now also offer paid "pro" versions, which if you are reading this guide you are unlikely to need). To install RStudio, go to:

https://www.rstudio.com/

Because RStudio's point is to be a user-friendly wrapper for R, its website is also a bit easier to navigate than the R directory. Click on the "Download" button in the black bar at the top of the screen, select the free "RStudio Desktop" version, then follow the instructions on the page to install the correct version for your operating system.

2.3 Navigating R Studio

Now, that you've installed R and RStudio, let's get oriented to the interface you'll be working in. When you open RStudio, you should get a window that looks roughly like this. You can see the screen is divided into four quadrants.

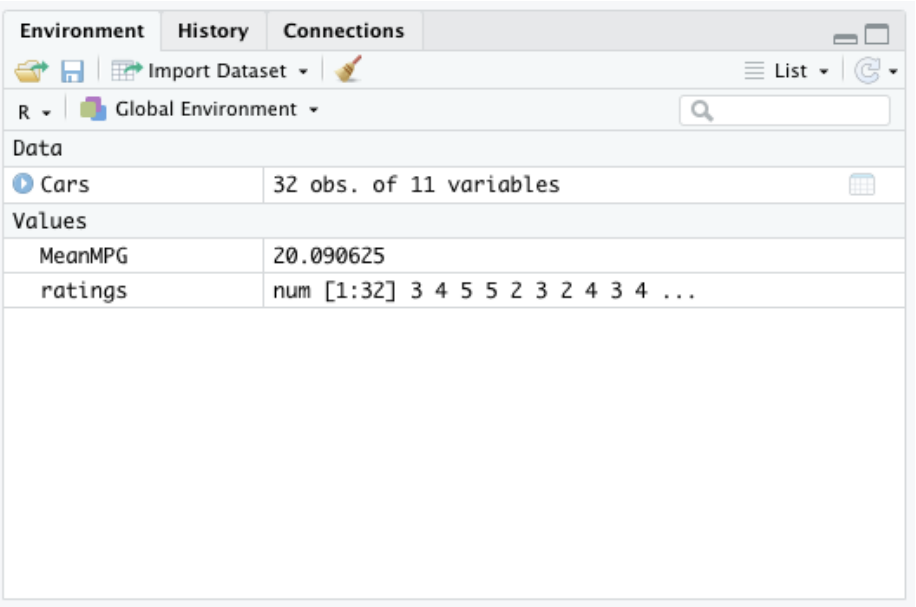


###Scripts The upper left corner is for your script[s]. This is where you write code for what you want R to do. R may not open a blank script for you automatically. To do so yourself, look in the top left corner, click on the white square with the plus sign in the green circle, and then select "R Script" from the dropdown menu. This will open a new blank script. You can also open a previously saved script using the button with a folder.

One of the main benefits of R is that you can write and save scripts that accomplish common tasks (cleaning data, doing a specific analysis, making graphs) and then use that same script on different data sets throughout your research career. This saves a lot of time and effort.

2.3.1 Global environment

The upper right corner is for your Global Environment. This displays the any other data (data frames, vectors, lists, or individual variables) that you save. While mostly we'll work with typed commands for R, there are some useful but-



tons in here.
In this example, you can see we have a data set called "Cars", as well as a vector called "ratings" and an individual variable called "MeanMPG" (we'll go over what all those terms mean later). One useful button is the blue arrow to the left of the data set's name ("Cars"). Clicking this will drop down a summary of that data set, including the column names, the type of data contained within each column, and the first few values in the data set. Another way to look at your data is by clicking the spreadsheet-shaped icon on the right side of the row. This will open a viewer, which lets you look at (but not edit) the full datas et as if it were in excel. If you want to start over, the broom at the top clears your environment, removing all the data from your current R session. (Importantly, this does NOT delete it from its source! You can always read that same data back in to R.)

2.3.2 View Pane

Below that, the bottom right pane is for viewing plots and for getting help in R. When you generate a plot using R, this is where it will show up, and where you can click to export or save it. It's also where the help interface is located. You can ask for help in r by typing a question mark before the name of any function, eg: `{r} ?sqrt()` You can also click into the help tab and use the search bar there to find help with a specific function.

2.3.3 Console

The bottom left quadrant is the console. In it, you can see the code you have run, the responses R generates, and any error messages. You can also type code directly into the console and run it from there.

Chapter 3

Basic commands

In this section we will cover basic commands in R.

3.1 R as a calculator

R is in many ways an interactive calculator, and practicing using R as such can help build intuitions for how to code in R. To practice using R as a calculator, try typing out the statements below (note: spacing does not matter in R).

You can 'run' these statements in a number of ways. The most efficient way is to click onto the line of code you want to run and press CMD + Return. You can also highlight multiple lines of code with your cursor and use these to execute all of the lines at once. The results will display in your Console (bottom panel in R Studio). (Alternatively: you can copy and paste any code we share into https://rdrr.io/snippets/ to generate the results in your browser).

```
2 + 2
```

```
4 - 1
```

```
5 * 5
```

```
10 / 2
```

3.2 Assigning values to variables

Another important feature of R is using variables. Variables are elements you can create and store in your R environment to use for future calculations.

Let's create a variable which refers to the number 7:

```
varA = 7
```

Note: we can also store variables which reflect other data types, a topic we will discuss ahead):

3.3 Using variables for calculations

With our variable 'a' saved, we can perform calculations by directly referencing varA. For instance, multiplying varA by 2 produces in Console produces "14".

```
varA * 2
```

Capitalization matters in R, so calculations with 'vara' will trigger an error since we have not defined this variable

```
vara * 2
```

We can flexibly use 'varA' for a wide variety of calculations.

```
varA + 20
varA - 3 / 2
70 / varA
varA * varA
```

3.4 Basic functions: sqrt()

Another important feature of R is 'functions'. Functions take inputs (numbers or variables) and transform them in different ways. To illustrate, lets apply a basic function, R's square root function, to a number, and then and our variable.

```
sqrt(4)
sqrt(varA)
```

We will return to functions in more detail in future chapters.

Chapter 4

Types of data in R

R can store all kinds of different data, varying in shape, class, and size. In this section, we'll first go over the various shapes of data available: variables, vectors, and data sets. Then we'll discuss the different classes of data: integer/numeric, character, logical, and factor. Finally we'll go over a few commands for discerning the size of your data.

4.1 Shape

4.1.1 Variables

You can store a single value with a command like:

```
x = 8
```

You can also store the output of a function as a variable, like:

```
y = sqrt(9)
```

or

```
z = mean(c(2, 4, 5, 3))
```

If you look in your global environment after running each of those lines of code, you'll see new entries under "Values" which list the variable name (x, y, and z) and the value stored (8, 3, and 3.5).

4.1.2 Vectors

You can also store strings of values, called vectors in R. You can string a set of values together using the **c()** command. Technically "c" stands for "concatenate" (a mathematical term for linking things together in a series), but in my mind I

always think of it as "combine". You can store strings of numbers, or, you can store strings of words/characters:

```
numbers = c(1, 5, 12, 7, 4)
words = c("good", "dog", "big", "fluffy", "treat")
```

There are some useful shortcuts for making certain types of vectors. If you want a vector made up of increasing or decreasing integers, you can use the : command, which works in increasing or decreasing order, like so:

```
1:7
```

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

```
12:6
```

```
## [1] 12 11 10  9  8  7  6
```

Another useful shortcut is the **rep()** function, short for "repeat." Here you enter first the variable or vector you want to repeat, and then how many times you want to repeat it. So if you needed to create a vector with the number 2 repeated five times, you'd write:

```
AllTwos = rep(2, 5)
AllTwos
```

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

You can also combine the c() and rep() commands to have a sequence repeat, like so:

```
PetDog = rep(c("Pet", "Dog"), 3)
PetDog
```

```
## [1] "Pet" "Dog" "Pet" "Dog" "Pet" "Dog"
```

4.1.2.1 Indexing vectors

Selecting one (or more) items out of the longer vector is called indexing. This is a very useful command which we'll build on in future sections. To index, square brackets after the vector to indicate the position of the value(s) you're interested in. So, to see the second item in the "numbers" vector, you would type **numbers[2]** and would expect to see the number 5 as the output, like so:

```
numbers[2]
```

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

You can also select multiple values in a vector, whether or not they are continuous.

```
# continuous, use : command
words[3:5]
```

```
## [1] "big"    "fluffy" "treat"
```

```
# non-continuous, use c() to create a list of the values you want to see
words[c(1,2,5)]
```

```
## [1] "good"  "dog"   "treat"
```

4.1.3 Data frames

Data can also be stored in a two-dimensional matrix called a data frame. Think of this like a spreadsheet or table of data. Here's an example of some data about cars that is already in the base R package:

```
mtcars
```

```
##                     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    4
## Mazda RX4 Wag      21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive     21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant            18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360         14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D          24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230           22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280           19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C          17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE         16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL         17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC        15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4  8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial  14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128           32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic        30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla     33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona      21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger   15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin        15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28         13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird   19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## 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
```

You'll learn more about how to get a descriptive sense of what's in a data frame in the next chapter. For now, it's worth noting that in addition to the actual numbers (or other types of data) comprising the table itself, each data frame can also have row names and column names. You can list or edit those names with the **rownames()** and **colnames()** commands.

```
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```
colnames(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

4.1.3.1 Indexing data frames

In the same way that you can index, or select, certain values out of a vector, you can also index the values you need out of a data frame. To do this, you use the following format: **dataframe[row, column]**. Here are some examples:

```
mtcars[4,1] #select the value in the fourth row and first column
```

```
## [1] 21.4
```

```
mtcars[2,6] #select the value in the second row and sixth column
```

```
## [1] 2.875
```

```
#You can select an entire row or column by leaving the other value blank:
mtcars[3,] #select the entire third row
```

```
##             mpg cyl disp hp drat   wt  qsec vs am gear carb
## Datsun 710 22.8   4  108 93 3.85 2.32 18.61  1  1    4    1
```

```
mtcars[,5] #select the entire fifth column
```

```
##  [1] 3.90 3.90 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 3.92 3.07 3.07 3.07 2.93
## [16] 3.00 3.23 4.08 4.93 4.22 3.70 2.76 3.15 3.73 3.08 4.08 4.43 3.77 4.22 3.62
## [31] 3.54 4.11
```

```
#You can also select multiple rows/columns by using a vector:
mtcars[1:5,] #select the contents of the first five rows
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

```
mtcars[12:16, 1:3] #within rows 12-16, select the first three columns
```

```
##                     mpg cyl  disp
## Merc 450SE         16.4   8 275.8
## Merc 450SL         17.3   8 275.8
## Merc 450SLC        15.2   8 275.8
## Cadillac Fleetwood 10.4   8 472.0
## Lincoln Continental 10.4   8 460.0
```

```
#Finally, you can also use the row and column names to select instead of the numbers. Note that y
mtcars[,"cyl"] #select all the values in the "cyl" column
```

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

```
mtcars["Honda Civic","mpg"] #select the miles per gallon (mpg) of the Honda Civic.
```

```
## [1] 30.4
```

```
mtcars[c("Hornet 4 Drive", "Hornet Sportabout"), ] #select all the columns for the Hornet 4 Drive
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

Often, you'll want to select a single column within a dataframe. The shortcut for this is **$**, so if you wanted to select the "cyl" column from mtcars, you could type:

```
mtcars$cyl
```

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

4.2 Class

There are three main types of vectors in R: numbers, characters, and logical. Numbers can be represented as **numeric** or **integer** class vectors depending on whether or not they have decimals, but those two classes function very similarly. Any string of words or letters can be stored as a **character** class vector. Finally,

logical vectors store data that has one of two values: **TRUE** or **FALSE**. You can find the out which type of data any vector is by using the **class()** argument.

```
#Let's look at the two vectors we created earlier:
numbers = c(1, 5, 12, 7, 4)
words = c("good", "dog", "big", "fluffy", "treat")

#What types of data do they contain?
class(numbers)
```

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

```
class(words)
```

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

```
#Note that each vector can only be one class of data. So if you create a vector that h
mixed = c("I am", 16, "going", "on", 17)
class(mixed)
```

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

```
#You can also see what class of vector a column within a data set has:
class(mtcars[,"mpg"])
```

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

You can create a logical vector either by combining TRUE or FALSE values with the **c()** function. For example:

```
#Creating a logical vector with c(). Note that you do not need quotes here because R r
logics = c(TRUE, FALSE, TRUE, T, F)
class(logics)
```

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

4.2.1 Relational Operators

Alternatively, you can create a logical variable by asking a question about an existing variable using a relational operator. These include **==, >, <, >=, and <=**. So for example, **x == y** asks the question, "is x equal to y?", while **x < y** asks the question "is x less than y?" Here, we apply those relational operators to our existing vectors:

```
#Creating a logical vector with c(). Note that you do not need quotes here because R r
numbers = c(1, 5, 12, 7, 4)
words = c("good", "dog", "big", "fluffy", "treat")
PetDog = rep(c("Pet", "Dog"), 3)

#We might ask which values in numbers are less than 10.
numbers < 10
```

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

```
#To save that as a logical variable saying which numbers are small, you could:
small = (numbers < 10)

#You can also use the == operator with character vectors:
PetDog == "Pet"
```

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

You can also ask a question about each value within a column of your data frame, and store the answers in a new column. For example, if you were curious which cars had relatively poor gas mileage (we'll operationalize that as getting less than 20 miles per gallon), you could do the following:

```
mtcars$badmileage = mtcars$mpg < 20
```

This creates a new column in our mtcars dataframe that record the TRUE/FALSE values for whether each car gets less than 20 miles per gallon.

```
mtcars[, c("mpg", "badmileage")] #look at the mpg and badmileage columns
```

```
##                      mpg badmileage
## Mazda RX4           21.0      FALSE
## Mazda RX4 Wag       21.0      FALSE
## Datsun 710          22.8      FALSE
## Hornet 4 Drive      21.4      FALSE
## Hornet Sportabout   18.7       TRUE
## Valiant             18.1       TRUE
## Duster 360          14.3       TRUE
## Merc 240D           24.4      FALSE
## Merc 230            22.8      FALSE
## Merc 280            19.2       TRUE
## Merc 280C           17.8       TRUE
## Merc 450SE          16.4       TRUE
## Merc 450SL          17.3       TRUE
## Merc 450SLC         15.2       TRUE
## Cadillac Fleetwood  10.4       TRUE
## Lincoln Continental 10.4       TRUE
## Chrysler Imperial   14.7       TRUE
## Fiat 128            32.4      FALSE
## Honda Civic         30.4      FALSE
## Toyota Corolla      33.9      FALSE
## Toyota Corona       21.5      FALSE
## Dodge Challenger    15.5       TRUE
## AMC Javelin         15.2       TRUE
## Camaro Z28          13.3       TRUE
```

```
## Pontiac Firebird    19.2       TRUE
## Fiat X1-9           27.3      FALSE
## Porsche 914-2       26.0      FALSE
## Lotus Europa        30.4      FALSE
## Ford Pantera L      15.8       TRUE
## Ferrari Dino        19.7       TRUE
## Maserati Bora       15.0       TRUE
## Volvo 142E          21.4      FALSE
```

```
class(mtcars$badmileage) #check the class of the new badmileage column
```

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

4.3 Size

There are a few useful commands for ascertaining the size of your data frame.

```
nrow(mtcars) # how many rows is mtcars?
```

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

```
ncol(mtcars) # how many columns is mtcars?
```

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

```
dim(mtcars) #print the row length and column length for mtcars.
```

```
## [1] 32 12
```

Chapter 5

Data Cleaning

In this section we will cover how to load data, clean your data, and wrangle your data in R.

5.1 Loading data into R

So you've got some data you would like to analyze. The first step is to load in your data.

For this task, we can typically use the read.csv command. Assuming our .csv data file is located in the same folder on our computer as our R file (this is always good practice!), we can simply use the following function which will store our data locally as the data frame "df":

```
df = read.csv("data.csv")

# If our csv happens to have multiple rows of headers (which is often the case when one is workin

#df = read.csv("data.csv", skip = 3, header = F)
```

5.2 Inspecting data in R

We can now inspect our data to ensure it looks ready for analysis. Ideally we have rows and columns that only reflect our data, and not some of info such as variable headers. To ensure everything looks good, we can use the following commands:

```
#df # this prints our entire data frame

#head(df) # 'head' shows a sneak peek of the first several rows and columns of our data
```

```
#names(df) # 'names' shows our variables names

# you can test out other commands as well, such as 'str', 'levels', etc)
```

5.3 Subsetting data

We might next want to subset our data on the basis of some variable of interest. For instance, perhaps we want to filter out subjects who behaved a certain way. To do this, we can use the subset function with some conditional logic.

```
# In R, if we want to check if a statement is true, we use "==", whereas we use "=" to
  a = 1 # assign value 1 to a
  b = 2 # assign value 2 to b
  a == 1 # check if a is equal to 1 (TRUE)
```

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

```
  a == 2 # check if a is equal to 2 (FALSE)
```

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

```
df_fair = subset(df, df$fairness_violation == "violator") # this creates a new datafra
```

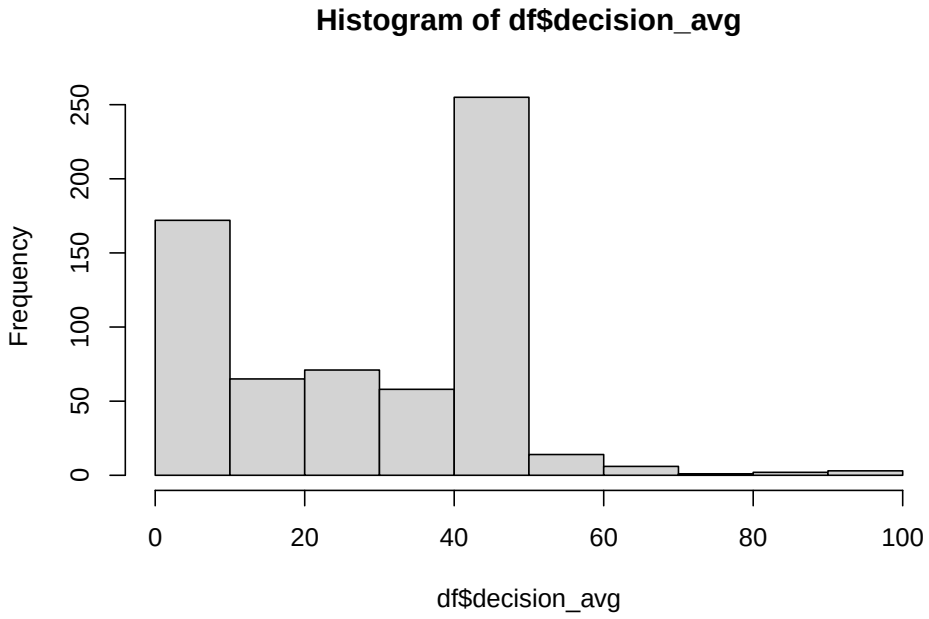
5.4 Computing averages of several variables

Another data cleaning task we might wish to accomplish is to create a variable which is the average of several variables. We can do this in a few different ways.

```
# Imagine we have five identical decisions made at different time points, if we want t

df$decision_avg = (df$decision1+df$decision2+df$decision3+df$decision4+df$decision5)/5

# We can now inspect the group average of this variable:

mean(df$decision_avg)
```

```
## [1] 30.97682
```

```
# We can also do a basic visualization (a histogram) of the data to ensure it looks ok

hist(df$decision_avg)
```



5.5 Recoding variables

A final task we might wish to accomplish is recoding our data. For instance, our raw data file might have gender coded as '1', '2', '3', and '4'. In R, we can easily update each value in this variable to reflect its true qualitative value.

```
head(df$gender)
```

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

```
# Currently, our gender variable is We can apply value labels to in

df$gender = factor(df$gender,
levels = c(1,2,3,4),
labels = c("Female", "Male", "Other", "Prefer Not To Say"))  # Here, each value is being changed

# Now we can observe this change:

head(df$gender)
```

```
## [1] Female Female Female Female Male   Female
## Levels: Female Male Other Prefer Not To Say
```

5.6 Removing missing data

Imagine we have missing values for subjects on a number of variables. If this variable is crucial to our analysis, we can exclude these subjects full stop. However, we might also wish to include their data when possible. Fortunately R has functions for both possibilities.

```
# We can use na.omit to remove any columns and rows with missing data

df = na.omit(df)

# Alternatively, we can selectively deploy na.omit within our actual statistical tests

t.test(df$decision1, df$decision2, na.omit=TRUE)
```

```
##
##  Welch Two Sample t-test
##
## data:  df$decision1 and df$decision2
## t = 0.37765, df = 1289.8, p-value = 0.7058
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.883128  2.780961
## sample estimates:
## mean of x mean of y
##  31.03715  30.58824
```

We hope this helps clarify how to perform some of the basic steps in data cleaning.