R Notebook

Chapter 2 R basics

In this book, we will be using the R software environment for all our analysis. You will learn R and data analysis techniques simultaneously. To follow along you will therefore need access to R. We also recommend the use of an integrated development environment (IDE), such as RStudio, to save your work. Note that it is common for a course or workshop to offer access to an R environment and an IDE through your web browser, as done by RStudio cloud. If you have access to such a resource, you don't need to install R and RStudio. However, if you intend on becoming an advanced data analyst, we highly recommend installing these tools on your computer. Both R and RStudio are free and available online.

2.1 Case study: US Gun Murders

Imagine you live in Europe and are offered a job in a US company with many locations across all states. It is a great job, but news with headlines such as US Gun Homicide Rate Higher Than Other Developed Countries have you worried. Charts like this may concern you even more: Or even worse, this version from everytown.org: But then you remember that the US is a large and diverse country with 50 very different states as well as the District of Columbia (DC). California, for example, has a larger population than Canada, and 20 US states have populations larger than that of Norway. In some respects, the variability across states in the US is akin to the

variability across countries in Europe. Furthermore, although not included in the charts above, the murder rates in Lithuania, Ukraine, and Russia are higher than 4 per 100,000. So perhaps the news reports that worried you are too superficial. You have options of where to live and want to determine the safety of each particular state. We will gain some insights by examining data related to gun homicides in the US during 2010 using R. Before we get started with our example, we need to cover logistics as well as some of the very basic building blocks that are required to gain more advanced R skills. Be aware that the usefulness of some of these building blocks may not be immediately obvious, but later in the book you will appreciate having mastered these skills.

2.2 The very basics

Before we get started with the motivating dataset, we need to cover the very basics of R.

2.2.1 Objects

Suppose a high school student asks us for help solving several quadratic equations of the form ax^2+bx+c = 0. The quadratic formula gives us the solutions: which of course change depending on the values of a, b, and c. One advantage of programming languages is that we can define variables and write expressions with these variables, similar to how we do so in math, but obtain a numeric solution. We will write out general code for the quadratic equation below, but if we are asked to solve x^2 + x -1 = 0, then we define:

```
a <- 1
b <- 1
c <- -1
```

which stores the values for later use. We use <- to assign values to

the variables.

We can also assign values using = instead of <-, but we recommend against using = to avoid confusion.

Copy and paste the code above into your console to define the three variables. Note that R does not print anything when we make this assignment. This means the objects were defined successfully. Had you made a mistake, you would have received an error message.

To see the value stored in a variable, we simply ask R to evaluate a and it shows the stored value:

```
a
```

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

A more explicit way to ask R to show us the value stored in a is using print like this:

```
print(a)
```

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

We use the term object to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, which are described later.

2.2.2 The workspace

As we define objects in the console, we are actually changing the workspace. You can see all the variables saved in your workspace by typing:

```
ls()
```

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

In RStudio, the Environment tab shows the values:

We should see a, b, and c. If you try to recover the value of a variable that is not in your workspace, you receive an error. For example, if you type x you will receive the following message: Error:

object ‘x’ not found.

```
# x
```

Now since these values are saved in variables, to obtain a solution to our equation, we use the quadratic formula:

```
(-b + sqrt(b^2 - 4*a*c) ) / ( 2*a )
## [1] 0.618034
(-b - sqrt(b^2 - 4*a*c) ) / ( 2*a )
## [1] -1.618034
```

2.2.3 Functions

Once you define variables, the data analysis process can usually be described as a series of functions applied to the data. R includes several predefined functions and most of the analysis pipelines we construct make extensive use of these.

We already used the install.packages, library, and ls functions. We also used the function sqrt to solve the quadratic equation above. There are many more prebuilt functions and even more can be added through packages. These functions do not appear in the workspace because you did not define them, but they are available for immediate use.

In general, we need to use parentheses to evaluate a function. If you type ls, the function is not evaluated and instead R shows you the code that defines the function. If you type ls() the function is evaluated and, as seen above, we see objects in the workspace.

Unlike ls, most functions require one or more arguments. Below is an example of how we assign an object to the argument of the function log. Remember that we earlier defined a to be 1:

```
log(8)
## [1] 2.079442
log(a)
```

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

You can find out what the function expects and what it does by reviewing the very useful manuals included in R. You can get help by using the help function like this:

```
help("log")
```

For most functions, we can also use this shorthand:

```
?log
```

The help page will show you what arguments the function is expecting. For example, log needs x and base to run. However, some arguments are required and others are optional. You can determine which arguments are optional by noting in the help document that a default value is assigned with =. Defining these is optional. For example, the base of the function log defaults to base = exp(1) making log the natural log by default.

If you want a quick look at the arguments without opening the help system, you can type:

```
args(log)
## function (x, base = exp(1))
## NULL
```

You can change the default values by simply assigning another object:

```
log(8, base = 2)
## [1] 3
```

Note that we have not been specifying the argument x as such:

```
log(x=8, base=2)
## [1] 3
```

The above code works, but we can save ourselves some typing: if no argument name is used, R assumes you are entering arguments in the order shown in the help file or by args. So by not using the names, it assumes the arguments are x followed by base:

```
log(8,2)
```
```
## [1] 3
```
If using the arguments' names, then we can include them in whatever order we want:

```
log(base = 2, x = 8)
```
```
## [1] 3
```
To specify arguments, we must use =, and cannot use <-.

There are some exceptions to the rule that functions need the parentheses to be evaluated. Among these, the most commonly used are the arithmetic and relational operators. For example:

```
2^3
```
```
## [1] 8
```
You can see the arithmetic operators by typing:

```
help("+")
```
or

```
?"+"
```
and the relational operators by typing:

```
help(">")
```
or

```
?">"
```
2.2.4 Other prebuilt objects

There are several datasets that are included for users to practice and test out functions. You can see all the available datasets by typing:

```
data()
```
This shows you the object name for these datasets. These datasets are objects that can be used by simply typing the name. For example, if you type:

```
co2
```

```
##          Jan    Feb    Mar    Apr    May    Jun
Jul    Aug    Sep    Oct
## 1959 315.42 316.31 316.50 317.56 318.13 318.00
316.39 314.65 313.68 313.18
## 1960 316.27 316.81 317.42 318.87 319.87 319.43
318.01 315.74 314.00 313.68
## 1961 316.73 317.54 318.38 319.31 320.42 319.61
318.42 316.63 314.83 315.16
## 1962 317.78 318.40 319.53 320.42 320.85 320.45
319.45 317.25 316.11 315.27
## 1963 318.58 318.92 319.70 321.22 322.08 321.31
319.58 317.61 316.05 315.83
## 1964 319.41 320.07 320.74 321.40 322.06 321.73
320.27 318.54 316.54 316.71
## 1965 319.27 320.28 320.73 321.97 322.00 321.71
321.05 318.71 317.66 317.14
## 1966 320.46 321.43 322.23 323.54 323.91 323.59
322.24 320.20 318.48 317.94
## 1967 322.17 322.34 322.88 324.25 324.83 323.93
322.38 320.76 319.10 319.24
## 1968 322.40 322.99 323.73 324.86 325.40 325.20
323.98 321.95 320.18 320.09
## 1969 323.83 324.26 325.47 326.50 327.21 326.54
325.72 323.50 322.22 321.62
## 1970 324.89 325.82 326.77 327.97 327.91 327.50
326.18 324.53 322.93 322.90
## 1971 326.01 326.51 327.01 327.62 328.76 328.40
327.20 325.27 323.20 323.40
## 1972 326.60 327.47 327.58 329.56 329.90 328.92
327.88 326.16 324.68 325.04
## 1973 328.37 329.40 330.14 331.33 332.31 331.90
330.70 329.15 327.35 327.02
## 1974 329.18 330.55 331.32 332.48 332.92 332.08
331.01 329.23 327.27 327.21
## 1975 330.23 331.25 331.87 333.14 333.80 333.43
331.73 329.90 328.40 328.17
```

```
## 1976 331.58 332.39 333.33 334.41 334.71 334.17
332.89 330.77 329.14 328.78
## 1977 332.75 333.24 334.53 335.90 336.57 336.10
334.76 332.59 331.42 330.98
## 1978 334.80 335.22 336.47 337.59 337.84 337.72
336.37 334.51 332.60 332.38
## 1979 336.05 336.59 337.79 338.71 339.30 339.12
337.56 335.92 333.75 333.70
## 1980 337.84 338.19 339.91 340.60 341.29 341.00
339.39 337.43 335.72 335.84
## 1981 339.06 340.30 341.21 342.33 342.74 342.08
340.32 338.26 336.52 336.68
## 1982 340.57 341.44 342.53 343.39 343.96 343.18
341.88 339.65 337.81 337.69
## 1983 341.20 342.35 342.93 344.77 345.58 345.14
343.81 342.21 339.69 339.82
## 1984 343.52 344.33 345.11 346.88 347.25 346.62
345.22 343.11 340.90 341.18
## 1985 344.79 345.82 347.25 348.17 348.74 348.07
346.38 344.51 342.92 342.62
## 1986 346.11 346.78 347.68 349.37 350.03 349.37
347.76 345.73 344.68 343.99
## 1987 347.84 348.29 349.23 350.80 351.66 351.07
349.33 347.92 346.27 346.18
## 1988 350.25 351.54 352.05 353.41 354.04 353.62
352.22 350.27 348.55 348.72
## 1989 352.60 352.92 353.53 355.26 355.52 354.97
353.75 351.52 349.64 349.83
## 1990 353.50 354.55 355.23 356.04 357.00 356.07
354.67 352.76 350.82 351.04
## 1991 354.59 355.63 357.03 358.48 359.22 358.12
356.06 353.92 352.05 352.11
## 1992 355.88 356.63 357.72 359.07 359.58 359.17
356.94 354.92 352.94 353.23
## 1993 356.63 357.10 358.32 359.41 360.23 359.55
357.53 355.48 353.67 353.95
```

```
## 1994 358.34 358.89 359.95 361.25 361.67 360.94
359.55 357.49 355.84 356.00
## 1995 359.98 361.03 361.66 363.48 363.82 363.30
361.94 359.50 358.11 357.80
## 1996 362.09 363.29 364.06 364.76 365.45 365.01
363.70 361.54 359.51 359.65
## 1997 363.23 364.06 364.61 366.40 366.84 365.68
364.52 362.57 360.24 360.83
##         Nov    Dec
## 1959 314.66 315.43
## 1960 314.84 316.03
## 1961 315.94 316.85
## 1962 316.53 317.53
## 1963 316.91 318.20
## 1964 317.53 318.55
## 1965 318.70 319.25
## 1966 319.63 320.87
## 1967 320.56 321.80
## 1968 321.16 322.74
## 1969 322.69 323.95
## 1970 323.85 324.96
## 1971 324.63 325.85
## 1972 326.34 327.39
## 1973 327.99 328.48
## 1974 328.29 329.41
## 1975 329.32 330.59
## 1976 330.14 331.52
## 1977 332.24 333.68
## 1978 333.75 334.78
## 1979 335.12 336.56
## 1980 336.93 338.04
## 1981 338.19 339.44
## 1982 339.09 340.32
## 1983 340.98 342.82
## 1984 342.80 344.04
## 1985 344.06 345.38
```

```
## 1986 345.48 346.72
## 1987 347.64 348.78
## 1988 349.91 351.18
## 1989 351.14 352.37
## 1990 352.69 354.07
## 1991 353.64 354.89
## 1992 354.09 355.33
## 1993 355.30 356.78
## 1994 357.59 359.05
## 1995 359.61 360.74
## 1996 360.80 362.38
## 1997 362.49 364.34
```

R will show you Mauna Loa atmospheric CO2 concentration data.

Other prebuilt objects are mathematical quantities, such as the constant π and ∞:

```
pi
## [1] 3.141593
Inf+1
## [1] Inf
```

2.2.5 Variable names

We have used the letters a, b, and c as variable names, but variable names can be almost anything. Some basic rules in R are that variable names have to start with a letter, can't contain spaces, and should not be variables that are predefined in R. For example, don't name one of your variables install.packages by typing something like install.packages <- 2.

A nice convention to follow is to use meaningful words that describe what is stored, use only lower case, and use underscores as a substitute for spaces. For the quadratic equations, we could use something like this:

```
solution_1 <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
solution_2 <- (-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

For more advice, we highly recommend studying Hadley Wickham's style guide.

2.2.6 Saving your workspace

Values remain in the workspace until you end your session or erase them with the function rm. But workspaces also can be saved for later use. In fact, when you quit R, the program asks you if you want to save your workspace. If you do save it, the next time you start R, the program will restore the workspace.

We actually recommend against saving the workspace this way because, as you start working on different projects, it will become harder to keep track of what is saved. Instead, we recommend you assign the workspace a specific name. You can do this by using the function save or save.image. To load, use the function load. When saving a workspace, we recommend the suffix rda or RData. In RStudio, you can also do this by navigating to the Session tab and choosing Save Workspace as. You can later load it using the Load Workspace options in the same tab. You can read the help pages on save, save.image, and load to learn more.

2.2.7 Motivating scripts

To solve another equation such as 3x^2 + 2x -1, we can copy and paste the code above and then redefine the variables and recompute the solution:

```
a <- 3
b <- 2
c <- -1
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
## [1] 0.3333333
(-b - sqrt(b^2 - 4*a*c)) / (2*a)
```

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

By creating and saving a script with the code above, we would not need to retype everything each time and, instead, simply change the variable names. Try writing the script above into an editor and notice how easy it is to change the variables and receive an answer.

2.2.8 Commenting your code

If a line of R code starts with the symbol #, it is not evaluated. We can use this to write reminders of why we wrote particular code. For example, in the script above we could add:

```
## Code to compute solution to quadratic equation of
the form ax^2 + bx + c
## define the variables
a <- 3
b <- 2
c <- -1

## now compute the solution
(-b + sqrt(b^2 - 4*a*c)) / (2*a)
## [1] 0.3333333
(-b - sqrt(b^2 - 4*a*c)) / (2*a)
## [1] -1
```

2.3 Exercises

1. What is the sum of the first 100 positive integers? The formula for the sum of integers 1 through n is n(n+1)/2. Define n=100 and then use R to compute the sum of 1 through 100 using the formula. What is the sum?

```
#number1 answer: 5050
n = 100
n*(n+1)/2
## [1] 5050
```

2. Now use the same formula to compute the sum of the integers

from 1 through 1,000.

```
#number2 answer: 500500
n=1000
n*(n+1)/2
## [1] 500500
```

3. Look at the result of typing the following code into R:

```
n <- 1000
x <- seq(1, n)
sum(x)
## [1] 500500
```

Based on the result, what do you think the functions seq and sum do? You can use help.

```
#number3 answer: b (seq creates a list of numbers and
sum adds them up)
```

4. In math and programming, we say that we evaluate a function when we replace the argument with a given number. So if we type sqrt(4), we evaluate the sqrt function. In R, you can evaluate a function inside another function. The evaluations happen from the inside out. Use one line of code to compute the log, in base 10, of the square root of 100.

```
#number4 answer: 1
log(sqrt(100),10)
## [1] 1
```

5. Which of the following will always return the numeric value stored in x? You can try out examples and use the help system if you want.

```
#number5 answer: c (log(exp(x)))
x <- seq(5, 10)
log(10^x)
## [1] 11.51293 13.81551 16.11810 18.42068 20.72327
23.02585
log10(x^10)
## [1]  6.989700  7.781513  8.450980  9.030900
9.542425 10.000000
```

```
log(exp(x))
## [1]  5  6  7  8  9 10
exp(log(x, base = 2))
## [1] 10.19531 13.26279 16.56604 20.08554 23.80570
27.71373
```

2.4 Data types

Variables in R can be of different types. For example, we need to distinguish numbers from character strings and tables from simple lists of numbers. The function class helps us determine what type of object we have:

```
a <- 2
class(a)
## [1] "numeric"
```

To work efficiently in R, it is important to learn the different types of variables and what we can do with these.

2.4.1 Data frames

Up to now, the variables we have defined are just one number. This is not very useful for storing data. The most common way of storing a dataset in R is in a data frame. Conceptually, we can think of a data frame as a table with rows representing observations and the different variables reported for each observation defining the columns. Data frames are particularly useful for datasets because we can combine different data types into one object.

A large proportion of data analysis challenges start with data stored in a data frame. For example, we stored the data for our motivating example in a data frame. You can access this dataset by loading the dslabs library and loading the murders dataset using the data function:

```
library(dslabs)
```

```
data(murders)
```
To see that this is in fact a data frame, we type:

```
class(murders)
```
```
## [1] "data.frame"
```
2.4.2 Examining an object

The function str is useful for finding out more about the structure of an object:

```
str(murders)
```
```
## 'data.frame':    51 obs. of  5 variables:
##  $ state     : chr  "Alabama" "Alaska" "Arizona"
"Arkansas" ...
##  $ abb       : chr  "AL" "AK" "AZ" "AR" ...
##  $ region    : Factor w/ 4 levels
"Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
##  $ population: num  4779736 710231 6392017 2915918
37253956 ...
##  $ total     : num  135 19 232 93 1257 ...
```
This tells us much more about the object. We see that the table has 51 rows (50 states plus DC) and five variables. We can show the first six lines using the function head:

```
head(murders)
```

In this dataset, each state is considered an observation and five variables are reported for each state.

Before we go any further in answering our original question about different states, let's learn more about the components of this object.

2.4.3 The accessor: $

For our analysis, we will need to access the different variables

represented by columns included in this data frame. To do this, we use the accessor operator $ in the following way:

```
murders$population
##  [1]  4779736   710231  6392017  2915918 37253956
5029196  3574097   897934
##  [9]   601723 19687653  9920000  1360301  1567582
12830632  6483802  3046355
## [17]  2853118  4339367  4533372  1328361  5773552
6547629  9883640  5303925
## [25]  2967297  5988927   989415  1826341  2700551
1316470  8791894  2059179
## [33] 19378102  9535483   672591 11536504  3751351
3831074 12702379  1052567
## [41]  4625364   814180  6346105 25145561  2763885
625741  8001024  6724540
## [49]  1852994  5686986   563626
```

But how did we know to use population? Previously, by applying the function str to the object murders, we revealed the names for each of the five variables stored in this table. We can quickly access the variable names using:

```
names(murders)
## [1] "state"      "abb"        "region"
"population" "total"
```

It is important to know that the order of the entries in murders$population preserves the order of the rows in our data table. This will later permit us to manipulate one variable based on the results of another. For example, we will be able to order the state names by the number of murders.

Tip: R comes with a very nice auto-complete functionality that saves us the trouble of typing out all the names. Try typing murders$p then hitting the tab key on your keyboard. This functionality and many other useful auto-complete features are available when working in

RStudio.

```
murders$population
##  [1]  4779736   710231  6392017  2915918 37253956
5029196  3574097   897934
##  [9]   601723 19687653  9920000  1360301  1567582
12830632  6483802  3046355
## [17]  2853118  4339367  4533372  1328361  5773552
6547629  9883640  5303925
## [25]  2967297  5988927   989415  1826341  2700551
1316470  8791894  2059179
## [33] 19378102  9535483   672591 11536504  3751351
3831074 12702379  1052567
## [41]  4625364   814180  6346105 25145561  2763885
625741  8001024  6724540
## [49]  1852994  5686986   563626
```

2.4.4 Vectors: numerics, characters, and logical

The object murders$population is not one number but several. We call these types of objects vectors. A single number is technically a vector of length 1, but in general we use the term vectors to refer to objects with several entries. The function length tells you how many entries are in the vector:

```
pop <- murders$population
length(pop)
## [1] 51
```

This particular vector is numeric since population sizes are numbers:

```
class(pop)
## [1] "numeric"
```

In a numeric vector, every entry must be a number.

To store character strings, vectors can also be of class character.

For example, the state names are characters:

```
class(murders$state)
## [1] "character"
```

As with numeric vectors, all entries in a character vector need to be a character.

Another important type of vectors are logical vectors. These must be either TRUE or FALSE.

```
z <- 3 == 2
z
## [1] FALSE
class(z)
## [1] "logical"
```

Here the == is a relational operator asking if 3 is equal to 2. In R, if you just use one =, you actually assign a variable, but if you use two == you test for equality.

You can see the other relational operators by typing:

```
?Comparison
```

In future sections, you will see how useful relational operators can be.

We discuss more important features of vectors after the next set of exercises.

Advanced: Mathematically, the values in pop are integers and there is an integer class in R. However, by default, numbers are assigned class numeric even when they are round integers. For example, class(1) returns numeric. You can turn them into class integer with the as.integer() function or by adding an L like this: 1L. Note the class by typing: class(1L)

```
class(1)
## [1] "numeric"
```

```
class(1L)
```
```
## [1] "integer"
```

2.4.5 Factors

In the murders dataset, we might expect the region to also be a character vector. However, it is not:

```
class(murders$region)
```
```
## [1] "factor"
```

It is a factor. Factors are useful for storing categorical data. We can see that there are only 4 regions by using the levels function:

```
levels(murders$region)
```
```
## [1] "Northeast"     "South"         "North Central"
"West"
```

In the background, R stores these levels as integers and keeps a map to keep track of the labels. This is more memory efficient than storing all the characters.

Note that the levels have an order that is different from the order of appearance in the factor object. The default in R is for the levels to follow alphabetical order. However, often we want the levels to follow a different order. You can specify an order through the levels argument when creating the factor with the factor function. For example, in the murders dataset regions are ordered from east to west. The function reorder lets us change the order of the levels of a factor variable based on a summary computed on a numeric vector. We will demonstrate this with a simple example, and will see more advanced ones in the Data Visualization part of the book.

Suppose we want the levels of the region by the total number of murders rather than alphabetical order. If there are values associated with each level, we can use the reorder and specify a data summary to determine the order. The following code takes the sum of the total murders in each region, and reorders the factor following these

sums.

```
region <- murders$region
value <- murders$total
region <- reorder(region, value, FUN = sum)
levels(region)
```

```
## [1] "Northeast"     "North Central" "West"
"South"
```

The new order is in agreement with the fact that the Northeast has the least murders and the South has the most.

Warning: Factors can be a source of confusion since sometimes they behave like characters and sometimes they do not. As a result, confusing factors and characters are a common source of bugs.

2.4.6 Lists

Data frames are a special case of lists. Lists are useful because you can store any combination of different types. You can create a list using the list function like this:

```
record <- list(name = "John Doe",
               student_id = 1234,
               grades = c(95, 82, 91, 97, 93),
               final_grade = "A")
```

The function c is described in Section 2.6.

This list includes a character, a number, a vector with five numbers, and another character.

```
record
```

```
## $name
## [1] "John Doe"
##
## $student_id
## [1] 1234
##
```

```
## $grades
## [1] 95 82 91 97 93
##
## $final_grade
## [1] "A"
```

```
class(record)
```

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

As with data frames, you can extract the components of a list with the accessor $.

```
record$student_id
```

```
## [1] 1234
```

We can also use double square brackets ([[) like this:

```
record[["student_id"]]
```

```
## [1] 1234
```

You should get used to the fact that in R, there are often several ways to do the same thing, such as accessing entries.

You might also encounter lists without variable names.

```
record2 <- list("John Doe", 1234)
record2
```

```
## [[1]]
## [1] "John Doe"
##
## [[2]]
## [1] 1234
```

If a list does not have names, you cannot extract the elements with $, but you can still use the brackets method and instead of providing the variable name, you provide the list index, like this:

```
record2[[1]]
```

```
## [1] "John Doe"
```

We won't be using lists until later, but you might encounter one in your own exploration of R. For this reason, we show you some

basics here.

2.4.7 Matrices

Matrices are another type of object that are common in R. Matrices are similar to data frames in that they are two-dimensional: they have rows and columns. However, like numeric, character and logical vectors, entries in matrices have to be all the same type. For this reason data frames are much more useful for storing data, since we can have characters, factors, and numbers in them.

Yet matrices have a major advantage over data frames: we can perform matrix algebra operations, a powerful type of mathematical technique. We do not describe these operations in this book, but much of what happens in the background when you perform a data analysis involves matrices. We cover matrices in more detail in Chapter 33.1 but describe them briefly here since some of the functions we will learn return matrices.

We can define a matrix using the matrix function. We need to specify the number of rows and columns.

```
mat <- matrix(1:12, 4, 3)
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

You can access specific entries in a matrix using square brackets ([]). If you want the second row, third column, you use:

```
mat[2, 3]
```

```
## [1] 10
```

If you want the entire second row, you leave the column spot empty:

```
mat[2, ]
## [1]  2  6 10
```
Notice that this returns a vector, not a matrix.

Similarly, if you want the entire third column, you leave the row spot empty:

```
mat[, 3]
## [1]  9 10 11 12
```
This is also a vector, not a matrix.

You can access more than one column or more than one row if you like. This will give you a new matrix.

```
mat[, 2:3]
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
## [3,]    7   11
## [4,]    8   12
```
You can subset both rows and columns:

```
mat[1:2, 2:3]
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
```
We can convert matrices into data frames using the function as.data.frame:

```
as.data.frame(mat)
```

You can also use single square brackets ([) to access rows and columns of a data frame:

```
data("murders")
murders[25, 1]
## [1] "Mississippi"
murders[2:3, ]
```

2.5 Exercises

1. Load the US murders dataset.

```
library(dslabs)
data(murders)
```

Use the function str to examine the structure of the murders object. Which of the following best describes the variables represented in this data frame?

```
#number1 answer: c (The state name, the abbreviation of the state name, the state's region, and the state's population and total number of murders for 2010.)
str(murders)
## 'data.frame':    51 obs. of  5 variables:
##  $ state     : chr  "Alabama" "Alaska" "Arizona" "Arkansas" ...
##  $ abb       : chr  "AL" "AK" "AZ" "AR" ...
##  $ region    : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
##  $ population: num  4779736 710231 6392017 2915918 37253956 ...
##  $ total     : num  135 19 232 93 1257 ...
```

2. What are the column names used by the data frame for these five variables?

```
#number2 answer: "state", "abb", "region", "population", "total"
names(murders)
## [1] "state"      "abb"        "region"     "population" "total"
```

3. Use the accessor $ to extract the state abbreviations and assign them to the object a. What is the class of this object?

```
#number3 answer: "character"
murders$state
##  [1] "Alabama"              "Alaska"
```

```
"Arizona"
##  [4] "Arkansas"              "California"
"Colorado"
##  [7] "Connecticut"           "Delaware"
"District of Columbia"
## [10] "Florida"               "Georgia"
"Hawaii"
## [13] "Idaho"                 "Illinois"
"Indiana"
## [16] "Iowa"                  "Kansas"
"Kentucky"
## [19] "Louisiana"             "Maine"
"Maryland"
## [22] "Massachusetts"         "Michigan"
"Minnesota"
## [25] "Mississippi"           "Missouri"
"Montana"
## [28] "Nebraska"              "Nevada"
"New Hampshire"
## [31] "New Jersey"            "New Mexico"
"New York"
## [34] "North Carolina"        "North Dakota"
"Ohio"
## [37] "Oklahoma"              "Oregon"
"Pennsylvania"
## [40] "Rhode Island"          "South Carolina"
"South Dakota"
## [43] "Tennessee"             "Texas"
"Utah"
## [46] "Vermont"               "Virginia"
"Washington"
## [49] "West Virginia"         "Wisconsin"
"Wyoming"
murders$abb
##  [1] "AL" "AK" "AZ" "AR" "CA" "CO" "CT" "DE" "DC"
"FL" "GA" "HI" "ID" "IL" "IN"
```

```
## [16] "IA" "KS" "KY" "LA" "ME" "MD" "MA" "MI" "MN"
"MS" "MO" "MT" "NE" "NV" "NH"
## [31] "NJ" "NM" "NY" "NC" "ND" "OH" "OK" "OR" "PA"
"RI" "SC" "SD" "TN" "TX" "UT"
## [46] "VT" "VA" "WA" "WV" "WI" "WY"
a <- murders$abb
class(a)
## [1] "character"
```

4. Now use the square brackets to extract the state abbreviations and assign them to the object b. Use the identical function to determine if a and b are the same.

```
#number4 answer: same
murders[["abb"]]
##  [1] "AL" "AK" "AZ" "AR" "CA" "CO" "CT" "DE" "DC"
"FL" "GA" "HI" "ID" "IL" "IN"
## [16] "IA" "KS" "KY" "LA" "ME" "MD" "MA" "MI" "MN"
"MS" "MO" "MT" "NE" "NV" "NH"
## [31] "NJ" "NM" "NY" "NC" "ND" "OH" "OK" "OR" "PA"
"RI" "SC" "SD" "TN" "TX" "UT"
## [46] "VT" "VA" "WA" "WV" "WI" "WY"
b <- murders[["abb"]]
q <- a==b
q
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE TRUE TRUE TRUE TRUE TRUE
## [16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE TRUE TRUE TRUE TRUE TRUE
## [31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE TRUE TRUE TRUE TRUE TRUE
## [46] TRUE TRUE TRUE TRUE TRUE TRUE
?identical
#use: identical(f,g)
identical(a,b)
## [1] TRUE
```

5. We saw that the region column stores a factor. You can corroborate this by typing:

```
class(murders$region)
## [1] "factor"
```

With one line of code, use the function levels and length to determine the number of regions defined by this dataset.

```
#number5 answer: length(levels(murders$region)) #> [1]
4
murders$region
##  [1] South         West          West          South
West
##  [6] West          Northeast     South         South
South
## [11] South         West          West          North
Central North Central
## [16] North Central North Central South         South
Northeast
## [21] South         Northeast     North Central North
Central South
## [26] North Central West          North Central West
Northeast
## [31] Northeast     West          Northeast     South
North Central
## [36] North Central South         West
Northeast     Northeast
## [41] South         North Central South         South
West
## [46] Northeast     South         West          South
North Central
## [51] West
## Levels: Northeast South North Central West
levels(murders$region)
## [1] "Northeast"     "South"         "North Central"
"West"
length(murders$region)
## [1] 51
length(levels(murders$region))
## [1] 4
```

```
levels(length(murders$region))
## NULL
```

6. The function table takes a vector and returns the frequency of each element. You can quickly see how many states are in each region by applying this function. Use this function in one line of code to create a table of states per region.

```
#number6 answer: table(murders$region)
library(dslabs)
data(murders)
?"table"
## Help on topic 'table' was found in the following
packages:
##
##   Package               Library
##   vctrs                 /Library/Frameworks/
R.framework/Versions/4.1/Resources/library
##   base                  /Library/Frameworks/
R.framework/Resources/library
##
##
## Using the first match ...
#usage: table(a)...
table(murders$region)
##
##     Northeast         South North Central
West
##             9            17            12
13
```

2.6 Vectors

In R, the most basic objects available to store data are vectors. As we have seen, complex datasets can usually be broken down into components that are vectors. For example, in a data frame, each column is a vector. Here we learn more about this important class.

2.6.1 Creating vectors

We can create vectors using the function c, which stands for concatenate. We use c to concatenate entries in the following way:

```
codes <- c(380, 124, 818)
codes
```

```
## [1] 380 124 818
```

We can also create character vectors. We use the quotes to denote that the entries are characters rather than variable names.

```
# country <- c("italy", "canada", "egypt")
```

In R you can also use single quotes:

```
# country <- c('italy', 'canada', 'egypt')
```

But be careful not to confuse the single quote ’ with the back quote `.

By now you should know that if you type:

```
# country <- c(italy, canada, egypt)
```

you receive an error because the variables italy, canada, and egypt are not defined. If we do not use the quotes, R looks for variables with those names and returns an error.

2.6.2 Names

Sometimes it is useful to name the entries of a vector. For example, when defining a vector of country codes, we can use the names to connect the two:

```
# codes <- c(italy = 380, canada = 124, egypt = 818)
codes
```

```
## [1] 380 124 818
```

The object codes continues to be a numeric vector:

```
class(codes)
```

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

but with names:

```
names(codes)
```

```
## NULL
```

If the use of strings without quotes looks confusing, know that you can use the quotes as well:

```
# codes <- c("italy" = 380, "canada" = 124, "egypt" =
818)
codes
```

```
## [1] 380 124 818
```

There is no difference between this function call and the previous one. This is one of the many ways in which R is quirky compared to other languages.

We can also assign names using the names functions:

```
codes <- c(380, 124, 818)
# country <- c("italy","canada","egypt")
# names(codes) <- country
codes
```

```
## [1] 380 124 818
```

2.6.3 Sequences

Another useful function for creating vectors generates sequences:

```
seq(1, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

The first argument defines the start, and the second defines the end which is included. The default is to go up in increments of 1, but a third argument lets us tell it how much to jump by:

```
seq(1, 10, 2)
```

```
## [1] 1 3 5 7 9
```

If we want consecutive integers, we can use the following shorthand:

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

When we use these functions, R produces integers, not numerics, because they are typically used to index something:

```
class(1:10)
## [1] "integer"
```

However, if we create a sequence including non-integers, the class changes:

```
class(seq(1, 10, 0.5))
## [1] "numeric"
```

2.6.4 Subsetting

We use square brackets to access specific elements of a vector. For the vector codes we defined above, we can access the second element using:

```
codes[2]
## [1] 124
```

You can get more than one entry by using a multi-entry vector as an index:

```
codes[c(1,3)]
## [1] 380 818
```

The sequences defined above are particularly useful if we want to access, say, the first two elements:

```
codes[1:2]
## [1] 380 124
```

If the elements have names, we can also access the entries using these names. Below are two examples.

```
codes["canada"]
## [1] NA
# codes[c("egypt","italy")]
```

2.7 Coercion

In general, coercion is an attempt by R to be flexible with data types. When an entry does not match the expected, some of the prebuilt R functions try to guess what was meant before throwing an error. This can also lead to confusion. Failing to understand coercion can drive programmers crazy when attempting to code in R since it behaves quite differently from most other languages in this regard. Let's learn about it with some examples.

We said that vectors must be all of the same type. So if we try to combine, say, numbers and characters, you might expect an error:

```
x <- c(1, "canada", 3)
```

But we don't get one, not even a warning! What happened? Look at x and its class:

```
x
## [1] "1"      "canada" "3"
class(x)
## [1] "character"
```

R coerced the data into characters. It guessed that because you put a character string in the vector, you meant the 1 and 3 to actually be character strings "1" and "3." The fact that not even a warning is issued is an example of how coercion can cause many unnoticed errors in R.

R also offers functions to change from one type to another. For example, you can turn numbers into characters with:

```
x <- 1:5
y <- as.character(x)
y
## [1] "1" "2" "3" "4" "5"
```

You can turn it back with as.numeric:

```
as.numeric(y)
## [1] 1 2 3 4 5
```

This function is actually quite useful since datasets that include numbers as character strings are common.

2.7.1 Not availables (NA)

When a function tries to coerce one type to another and encounters an impossible case, it usually gives us a warning and turns the entry into a special value called an NA for “not available.” For example:

```
x <- c("1", "b", "3")
as.numeric(x)
## Warning: NAs introduced by coercion
## [1]  1 NA  3
```

R does not have any guesses for what number you want when you type b, so it does not try.

As a data scientist you will encounter the NAs often as they are generally used for missing data, a common problem in real-world datasets.

2.8 Exercises

1. Use the function c to create a vector with the average high temperatures in January for Beijing, Lagos, Paris, Rio de Janeiro, San Juan, and Toronto, which are 35, 88, 42, 84, 81, and 30 degrees Fahrenheit. Call the object temp.

```
#number1 answer:
temp <- c(35, 88, 42, 84, 81, 30)
temp
## [1] 35 88 42 84 81 30
```

2. Now create a vector with the city names and call the object city.

```
#number2 answer:
city <- c('Beijing', 'Lagos', 'Paris', 'Rio de
Janeiro', 'San Juan', 'Toronto')
```

```
city
## [1] "Beijing"        "Lagos"          "Paris"
"Rio de Janeiro"
## [5] "San Juan"       "Toronto"
```

3. Use the names function and the objects defined in the previous exercises to associate the temperature data with its corresponding city.

```
#number3 answer:
temp <- c(35, 88, 42, 84, 81, 30)
city <- c('Beijing', 'Lagos', 'Paris', 'Rio de
Janeiro', 'San Juan', 'Toronto')
names(temp) <- city
temp
##        Beijing          Lagos          Paris Rio de
Janeiro       San Juan
##             35             88             42
84             81
##        Toronto
##             30
```

4. Use the [and : operators to access the temperature of the first three cities on the list.

```
#number4 answer:
temp[1:3]
## Beijing   Lagos   Paris
##      35      88      42
```

5. Use the [operator to access the temperature of Paris and San Juan.

```
#number5 answer:
temp[c(3,5)]
##    Paris San Juan
##       42       81
```

6. Use the : operator to create a sequence of numbers 12, 13, 14, ..., 73.

```
#number6 answer:
12:73
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36
## [26] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61
## [51] 62 63 64 65 66 67 68 69 70 71 72 73
#seq(12,73) is same.
```

7. Create a vector containing all the positive odd numbers smaller than 100.

```
#number7 answer:
seq(1,100,2)
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31
33 35 37 39 41 43 45 47 49
## [26] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81
83 85 87 89 91 93 95 97 99
#seq(1,99,2) is same.
```

8. Create a vector of numbers that starts at 6, does not pass 55, and adds numbers in increments of 4/7: 6, 6 + 4/7, 6 + 8/7, and so on. How many numbers does the list have? Hint: use seq and length.

```
#number8 answer: 86
seq(6,55,4/7)
##  [1]  6.000000  6.571429  7.142857  7.714286
8.285714  8.857143  9.428571
##  [8] 10.000000 10.571429 11.142857 11.714286
12.285714 12.857143 13.428571
## [15] 14.000000 14.571429 15.142857 15.714286
16.285714 16.857143 17.428571
## [22] 18.000000 18.571429 19.142857 19.714286
20.285714 20.857143 21.428571
## [29] 22.000000 22.571429 23.142857 23.714286
24.285714 24.857143 25.428571
## [36] 26.000000 26.571429 27.142857 27.714286
28.285714 28.857143 29.428571
## [43] 30.000000 30.571429 31.142857 31.714286
32.285714 32.857143 33.428571
## [50] 34.000000 34.571429 35.142857 35.714286
```

```
36.285714 36.857143 37.428571
## [57] 38.000000 38.571429 39.142857 39.714286
40.285714 40.857143 41.428571
## [64] 42.000000 42.571429 43.142857 43.714286
44.285714 44.857143 45.428571
## [71] 46.000000 46.571429 47.142857 47.714286
48.285714 48.857143 49.428571
## [78] 50.000000 50.571429 51.142857 51.714286
52.285714 52.857143 53.428571
## [85] 54.000000 54.571429
length(seq(6,55,4/7))
## [1] 86
```

9. What is the class of the following object a <- seq(1, 10, 0.5)?

```
#number9 answer: "numeric"
a <- seq(1,10,0.5)
class(a)
## [1] "numeric"
```

10. What is the class of the following object a <- seq(1, 10)?

```
#number10 answer: "interger"
a <- seq(1,10)
class(a)
## [1] "integer"
```

11. The class of class(a<-1) is numeric, not integer. R defaults to numeric and to force an integer, you need to add the letter L. Confirm that the class of 1L is integer.

```
#number11 answer: "interger"
class(1L)
## [1] "integer"
```

12. Define the following vector:and coerce it to get integers.

```
#number12 answer:
x <- c("1", "3", "5")
class(x)
## [1] "character"
as.numeric(x)
## [1] 1 3 5
class(as.numeric(x))
```

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

2.9 Sorting

Now that we have mastered some basic R knowledge, let's try to gain some insights into the safety of different states in the context of gun murders.

2.9.1 sort

Say we want to rank the states from least to most gun murders. The function sort sorts a vector in increasing order. We can therefore see the largest number of gun murders by typing:

```
library(dslabs)
data(murders)
sort(murders$total)
```

```
##  [1]    2    4    5    5    7    8   11   12   12
16   19   21   22   27   32
## [16]   36   38   53   63   65   67   84   93   93
97   97   99  111  116  118
## [31]  120  135  142  207  219  232  246  250  286
293  310  321  351  364  376
## [46]  413  457  517  669  805 1257
```

However, this does not give us information about which states have which murder totals. For example, we don't know which state had 1257.

2.9.2 order

The function order is closer to what we want. It takes a vector as input and returns the vector of indexes that sorts the input vector. This may sound confusing so let's look at a simple example. We can create a vector and sort it:

```
x <- c(31, 4, 15, 92, 65)
```

```
sort(x)
```
```
## [1]  4 15 31 65 92
```
Rather than sort the input vector, the function order returns the index that sorts input vector:

```
index <- order(x)
x[index]
```
```
## [1]  4 15 31 65 92
```
This is the same output as that returned by sort(x). If we look at this index, we see why it works:

```
x
```
```
## [1] 31  4 15 92 65
```
```
order(x)
```
```
## [1] 2 3 1 5 4
```
The second entry of x is the smallest, so order(x) starts with 2. The next smallest is the third entry, so the second entry is 3 and so on.

How does this help us order the states by murders? First, remember that the entries of vectors you access with $ follow the same order as the rows in the table. For example, these two vectors containing state names and abbreviations, respectively, are matched by their order:

```
murders$state[1:6]
```
```
## [1] "Alabama"    "Alaska"     "Arizona"
"Arkansas"   "California"
## [6] "Colorado"
```
```
murders$abb[1:6]
```
```
## [1] "AL" "AK" "AZ" "AR" "CA" "CO"
```
This means we can order the state names by their total murders. We first obtain the index that orders the vectors according to murder totals and then index the state names vector:

```
ind <- order(murders$total)
murders$abb[ind]
```
```
##  [1] "VT" "ND" "NH" "WY" "HI" "SD" "ME" "ID" "MT"
```

```
"RI" "AK" "IA" "UT" "WV" "NE"
## [16] "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA"
"CT" "WI" "DC" "OK" "KY" "MA"
## [31] "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC"
"MD" "OH" "MO" "LA" "IL" "GA"
## [46] "MI" "PA" "NY" "FL" "TX" "CA"
```

According to the above, California had the most murders.

2.9.3 max and which.max

If we are only interested in the entry with the largest value, we can use max for the value:

```
max(murders$total)
## [1] 1257
```

and which.max for the index of the largest value:

```
i_max <- which.max(murders$total)
murders$state[i_max]
## [1] "California"
```

For the minimum, we can use min and which.min in the same way.

Does this mean California is the most dangerous state? In an upcoming section, we argue that we should be considering rates instead of totals. Before doing that, we introduce one last order-related function: rank.

2.9.4 rank

Although not as frequently used as order and sort, the function rank is also related to order and can be useful. For any given vector it returns a vector with the rank of the first entry, second entry, etc., of the input vector. Here is a simple example:

```
x <- c(31, 4, 15, 92, 65)
rank(x)
## [1] 3 1 2 5 4
```

To summarize, let's look at the results of the three functions we have introduced:

2.9.5 Beware of recycling

Another common source of unnoticed errors in R is the use of recycling. We saw that vectors are added elementwise. So if the vectors don't match in length, it is natural to assume that we should get an error. But we don't. Notice what happens:

```
x <- c(1, 2, 3)
y <- c(10, 20, 30, 40, 50, 60, 70)
x+y
## Warning in x + y: longer object length is not a
multiple of shorter object
## length
## [1] 11 22 33 41 52 63 71
```

We do get a warning, but no error. For the output, R has recycled the numbers in x. Notice the last digit of numbers in the output.

2.10 Exercises

For these exercises we will use the US murders dataset. Make sure you load it prior to starting.

```
library(dslabs)
data("murders")
```

1. Use the $ operator to access the population size data and store it as the object pop. Then use the sort function to redefine pop so that it is sorted. Finally, use the [operator to report the smallest population size.

```
#number1 answer:
murders$population
##  [1]  4779736   710231  6392017  2915918 37253956
5029196  3574097   897934
```

```
##  [9]   601723 19687653  9920000  1360301  1567582
12830632  6483802  3046355
## [17]  2853118  4339367  4533372  1328361  5773552
6547629  9883640  5303925
## [25]  2967297  5988927   989415  1826341  2700551
1316470  8791894  2059179
## [33] 19378102  9535483   672591 11536504  3751351
3831074 12702379  1052567
## [41]  4625364   814180  6346105 25145561  2763885
625741  8001024  6724540
## [49]  1852994  5686986   563626
```

```
pop <- murders$population
sort(pop)
```

```
##  [1]   563626   601723   625741   672591   710231
814180   897934   989415
##  [9]  1052567  1316470  1328361  1360301  1567582
1826341  1852994  2059179
## [17]  2700551  2763885  2853118  2915918  2967297
3046355  3574097  3751351
## [25]  3831074  4339367  4533372  4625364  4779736
5029196  5303925  5686986
## [33]  5773552  5988927  6346105  6392017  6483802
6547629  6724540  8001024
## [41]  8791894  9535483  9883640  9920000 11536504
12702379 12830632 19378102
## [49] 19687653 25145561 37253956
```

```
min(murders$population)
```

```
## [1] 563626
```

2. Now instead of the smallest population size, find the index of the entry with the smallest population size. Hint: use order instead of sort.

```
#number2 answer: 51
order(pop)
```

```
##  [1] 51  9 46 35  2 42  8 27 40 30 20 12 13 28 49 32
29 45 17  4 25 16  7 37 38
## [26] 18 19 41  1  6 24 50 21 26 43  3 15 22 48 47 31
```

```
34 23 11 36 39 14 33 10 44
## [51]  5
```

3. We can actually perform the same operation as in the previous exercise using the function which.min. Write one line of code that does this.

```
#number3 answer:
which.min(murders$population)
## [1] 51
```

4. Now we know how small the smallest state is and we know which row represents it. Which state is it? Define a variable states to be the state names from the murders data frame. Report the name of the state with the smallest population.

```
#number4 answer: "Wyoming"
states <- murders$state
#murder$state[which.min(murders$population)]
states[which.min(murders$population)]
## [1] "Wyoming"
```

5. You can create a data frame using the data.frame function. Here is a quick example:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de
Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature =
temp)
```

Use the rank function to determine the population rank of each state from smallest population size to biggest. Save these ranks in an object called ranks, then create a data frame with the state name and its rank. Call the data frame my_df.

```
#number5 answer:
rank(murders$population)
##  [1] 29  5 36 20 51 30 23  7  2 49 44 12 13 47 37 22
19 26 27 11 33 38 43 31 21
## [26] 34  8 14 17 10 41 16 48 42  4 45 24 25 46  9 28
```

```
6 35 50 18  3 40 39 15 32
## [51]  1
ranks <- rank(murders$population)
my_df <- data.frame(name=murders$state, ranks)
data.frame(name=murders$state, ranks)
```

6. Repeat the previous exercise, but this time order my_df so that the states are ordered from least populous to most populous. Hint: create an object ind that stores the indexes needed to order the population values. Then use the bracket operator [to re-order each column in the data frame.

```
#number6 answer:
ind <- order(murders$population)
states <- murders$state
#name=murders$state[ind]=states[ind] / ranks[ind]
my_df <- data.frame(states[ind], ranks[ind])
data.frame(states[ind], ranks[ind])
```

7. The na_example vector represents a series of counts. You can quickly examine the object using:

```
data("na_example")
str(na_example)
##  int [1:1000] 2 1 3 2 1 3 1 4 3 2 ...
```

However, when we compute the average with the function mean, we obtain an NA:

```
mean(na_example)
## [1] NA
```

The is.na function returns a logical vector that tells us which entries are NA. Assign this logical vector to an object called ind and determine how many NAs does na_example have.

```
#number7 answer: 145
ind <- is.na(na_example)
sum(ind)
```

```
## [1] 145
#sum(na_example) #> [NA]
#sum(sort(murders$population)) #> [1] 309864228
#sum(function...) is available.
```

8. Now compute the average again, but only for the entries that are not NA. Hint: remember the ! operator.

```
#number8 answer: 2.301754
?"!"
#! indicates logical negation (NOT).
sum(!ind)
## [1] 855
mean(na_example)
## [1] NA
#mean(na_example(!ind)) is not correct shape.
mean(na_example[!ind])
## [1] 2.301754
```

2.11 Vector arithmetics

California had the most murders, but does this mean it is the most dangerous state? What if it just has many more people than any other state? We can quickly confirm that California indeed has the largest population:

```
library(dslabs)
data("murders")
murders$state[which.max(murders$population)]
## [1] "California"
```

with over 37 million inhabitants. It is therefore unfair to compare the totals if we are interested in learning how safe the state is. What we really should be computing is the murders per capita. The reports we describe in the motivating section used murders per 100,000 as the unit. To compute this quantity, the powerful vector arithmetic capabilities of R come in handy.

2.11.1 Rescaling a vector

In R, arithmetic operations on vectors occur element-wise. For a quick example, suppose we have height in inches:

```
inches <- c(69, 62, 66, 70, 70, 73, 67, 73, 67, 70)
```

and want to convert to centimeters. Notice what happens when we multiply inches by 2.54:

```
inches * 2.54
##  [1] 175.26 157.48 167.64 177.80 177.80 185.42
170.18 185.42 170.18 177.80
```

In the line above, we multiplied each element by 2.54. Similarly, if for each entry we want to compute how many inches taller or shorter than 69 inches, the average height for males, we can subtract it from every entry like this:

```
inches - 69
##  [1]  0 -7 -3  1  1  4 -2  4 -2  1
```

2.11.2 Two vectors

If we have two vectors of the same length, and we sum them in R, they will be added entry by entry as follows:

The same holds for other mathematical operations, such as -, * and /.

This implies that to compute the murder rates we can simply type:

```
murder_rate <- murders$total / murders$population *
100000
```

Once we do this, we notice that California is no longer near the top of the list. In fact, we can use what we have learned to order the states by murder rate:

```
murders$abb[order(murder_rate)]
##  [1] "VT" "NH" "HI" "ND" "IA" "ID" "UT" "ME" "WY"
```

```
"OR" "SD" "MN" "MT" "CO" "WA"
## [16] "WV" "RI" "WI" "NE" "MA" "IN" "KS" "NY" "KY"
"AK" "OH" "CT" "NJ" "AL" "IL"
## [31] "OK" "NC" "NV" "VA" "AR" "TX" "NM" "CA" "FL"
"TN" "PA" "AZ" "GA" "MS" "MI"
## [46] "DE" "SC" "MD" "MO" "LA" "DC"
```

2.12 Exercises

1. Previously we created this data frame:

```
temp <- c(35, 88, 42, 84, 81, 30)
city <- c("Beijing", "Lagos", "Paris", "Rio de
Janeiro",
          "San Juan", "Toronto")
city_temps <- data.frame(name = city, temperature =
temp)
```

Remake the data frame using the code above, but add a line that converts the temperature from Fahrenheit to Celsius. The conversion is C = (F - 32).

```
#number1 answer:
#temp <- c(35, 88, 42, 84, 81, 30)
temp0 <- (temp-32)*5/9
temp0
## [1]  1.666667 31.111111  5.555556 28.888889
27.222222 -1.111111
city <- c("Beijing", "Lagos", "Paris", "Rio de
Janeiro",
          "San Juan", "Toronto")
city_temps0 <- data.frame(name = city, temperature =
temp0)
```

2. What is the following sum 1+1/2^2 + 1/3^2 + 1/100^2? Hint: thanks to Euler, we know it should be close to ^2/6.

```
#number2 answer: 1.634984
#1/seq(1,100)
sum(1/seq(1,100)^2)
```

```
## [1] 1.634984
```

3. Compute the per 100,000 murder rate for each state and store it in the object murder_rate. Then compute the average murder rate for the US using the function mean. What is the average?

```
#number3 answer:2.779125
murder_rate <- murders$total / murders$population *
100000
mean(murder_rate)
```

```
## [1] 2.779125
```

2.13 Indexing

R provides a powerful and convenient way of indexing vectors. We can, for example, subset a vector based on properties of another vector. In this section, we continue working with our US murders example, which we can load like this:

```
library(dslabs)
data("murders")
```

2.13.1 Subsetting with logicals

We have now calculated the murder rate using:

```
murder_rate <- murders$total / murders$population *
100000
```

Imagine you are moving from Italy where, according to an ABC news report, the murder rate is only 0.71 per 100,000. You would prefer to move to a state with a similar murder rate. Another powerful feature of R is that we can use logicals to index vectors. If we compare a vector to a single number, it actually performs the test for each entry. The following is an example related to the question above:

```
ind <- murder_rate < 0.71
```

If we instead want to know if a value is less or equal, we can use:

```
ind <- murder_rate <= 0.71
```

Note that we get back a logical vector with TRUE for each entry smaller than or equal to 0.71. To see which states these are, we can leverage the fact that vectors can be indexed with logicals.

```
murders$state[ind]
## [1] "Hawaii"        "Iowa"          "New Hampshire"
"North Dakota"
## [5] "Vermont"
```

In order to count how many are TRUE, the function sum returns the sum of the entries of a vector and logical vectors get coerced to numeric with TRUE coded as 1 and FALSE as 0. Thus we can count the states using:

```
sum(ind)
## [1] 5
```

2.13.2 Logical operators

Suppose we like the mountains and we want to move to a safe state in the western region of the country. We want the murder rate to be at most 1. In this case, we want two different things to be true. Here we can use the logical operator and, which in R is represented with &. This operation results in TRUE only when both logicals are TRUE. To see this, consider this example:

```
#TRUE must be capital letter
TRUE&TRUE
## [1] TRUE
TRUE&FALSE
## [1] FALSE
FALSE&FALSE
## [1] FALSE
```

For our example, we can form two logicals:

```
west <- murders$region == "West"
safe <- murder_rate <= 1
```

and we can use the & to get a vector of logicals that tells us which

states satisfy both conditions:

```
ind <- safe & west
murders$state[ind]
## [1] "Hawaii"  "Idaho"   "Oregon"  "Utah"
"Wyoming"
```

2.13.3 which

Suppose we want to look up California's murder rate. For this type of operation, it is convenient to convert vectors of logicals into indexes instead of keeping long vectors of logicals. The function which tells us which entries of a logical vector are TRUE. So we can type:

```
ind <- which(murders$state == "California")
murder_rate[ind]
## [1] 3.374138
#murder_rate <- murders$total / murders$population *
100000
```

2.13.4 match

If instead of just one state we want to find out the murder rates for several states, say New York, Florida, and Texas, we can use the function match. This function tells us which indexes of a second vector match each of the entries of a first vector:

```
ind <- match(c("New York", "Florida", "Texas"),
murders$state)
ind
## [1] 33 10 44
```

Now we can look at the murder rates:

```
murder_rate[ind]
## [1] 2.667960 3.398069 3.201360
```

2.13.5 %in%

If rather than an index we want a logical that tells us whether or not

each element of a first vector is in a second, we can use the function %in%. Let's imagine you are not sure if Boston, Dakota, and Washington are states. You can find out like this:

```
c("Boston", "Dakota", "Washington") %in% murders$state
## [1] FALSE FALSE  TRUE
```

Note that we will be using %in% often throughout the book.

Advanced: There is a connection between match and %in% through which. To see this, notice that the following two lines produce the same index (although in different order):

```
match(c("New York", "Florida", "Texas"), murders$state)
## [1] 33 10 44
which(murders$state%in%c("New York", "Florida",
"Texas"))
## [1] 10 33 44
#order is important?
#which(c("New York", "Florida", "Texas")
%in%murders$state) #> [1] 1 2 3
#which(FALSE, FALSE, TRUE) #> integer(0)
```

2.14 Exercises

Start by loading the library and data.

```
library(dslabs)
data(murders)
```

1. Compute the per 100,000 murder rate for each state and store it in an object called murder_rate. Then use logical operators to create a logical vector named low that tells us which entries of murder_rate are lower than 1.

```
#number1 answer:
murder_rate <- murders$total / murders$population *
100000
low <- murder_rate < 1
```

2. Now use the results from the previous exercise and the

function which to determine the indices of murder_rate associated with values lower than 1.

```
#number2 answer:
which(low)
##  [1] 12 13 16 20 24 30 35 38 42 45 46 51
```

3. Use the results from the previous exercise to report the names of the states with murder rates lower than 1.

```
#number3 answer:
murders$state[low]
##  [1] "Hawaii"        "Idaho"         "Iowa"
"Maine"
##  [5] "Minnesota"     "New Hampshire" "North Dakota"
"Oregon"
##  [9] "South Dakota"  "Utah"          "Vermont"
"Wyoming"
```

4. Now extend the code from exercises 2 and 3 to report the states in the Northeast with murder rates lower than 1. Hint: use the previously defined logical vector low and the logical operator &.

```
#number4 answer: "Maine" "New Hampshire" "Vermont"
#low <- murder_rate < 1
Northeast <- murders$region == "Northeast"
ind <- Northeast & low
murders$state[ind]
## [1] "Maine"         "New Hampshire" "Vermont"
```

5. In a previous exercise we computed the murder rate for each state and the average of these numbers. How many states are below the average?

```
#number5 answer: 27
mean(murder_rate)
## [1] 2.779125
#murder_rate < mean(murder_rate)
murders$state[murder_rate < mean(murder_rate)]
##  [1] "Alaska"        "Colorado"      "Connecticut"
"Hawaii"
```

```
##  [5] "Idaho"         "Indiana"       "Iowa"
"Kansas"
##  [9] "Kentucky"      "Maine"         "Massachusetts"
"Minnesota"
## [13] "Montana"       "Nebraska"      "New Hampshire"
"New York"
## [17] "North Dakota"  "Ohio"          "Oregon"
"Rhode Island"
## [21] "South Dakota"  "Utah"          "Vermont"
"Washington"
## [25] "West Virginia" "Wisconsin"     "Wyoming"
sum(murder_rate < mean(murder_rate))
## [1] 27
```

6. Use the match function to identify the states with abbreviations AK, MI, and IA. Hint: start by defining an index of the entries of murders$abb that match the three abbreviations, then use the [operator to extract the states.

```
#number6 answer: "Alaska" "Michigan" "Iowa"
match(c("AK", "MI", "IA"), murders$abb)
## [1]  2 23 16
murders$state[match(c("AK", "MI", "IA"), murders$abb)]
## [1] "Alaska"   "Michigan" "Iowa"
murders$stat[c(2, 23, 16)]
## [1] "Alaska"   "Michigan" "Iowa"
#murders$state[match(c("AK", "MI", "IA"), murders$abb)]
= murders$stat[c(2, 23, 16)]
```

7. Use the %in% operator to create a logical vector that answers the question: which of the following are actual abbreviations: MA, ME, MI, MO, MU?

```
#number7 answer: MA, ME, MI, MO
c("MA", "ME", "MI", "MO", "MU") %in% murders$abb
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
```

8. Extend the code you used in exercise 7 to report the one entry that is not an actual abbreviation. Hint: use the ! operator, which turns FALSE into TRUE and vice versa, then which to

obtain an index.

```
#number8 answer:
qq <- c("MA", "ME", "MI", "MO", "MU") %in% murders$abb
!qq
## [1] FALSE FALSE FALSE FALSE  TRUE
which(!qq)
## [1] 5
c("MA", "ME", "MI", "MO", "MU")[5]
## [1] "MU"
c("MA", "ME", "MI", "MO", "MU")[which(!qq)]
## [1] "MU"
```

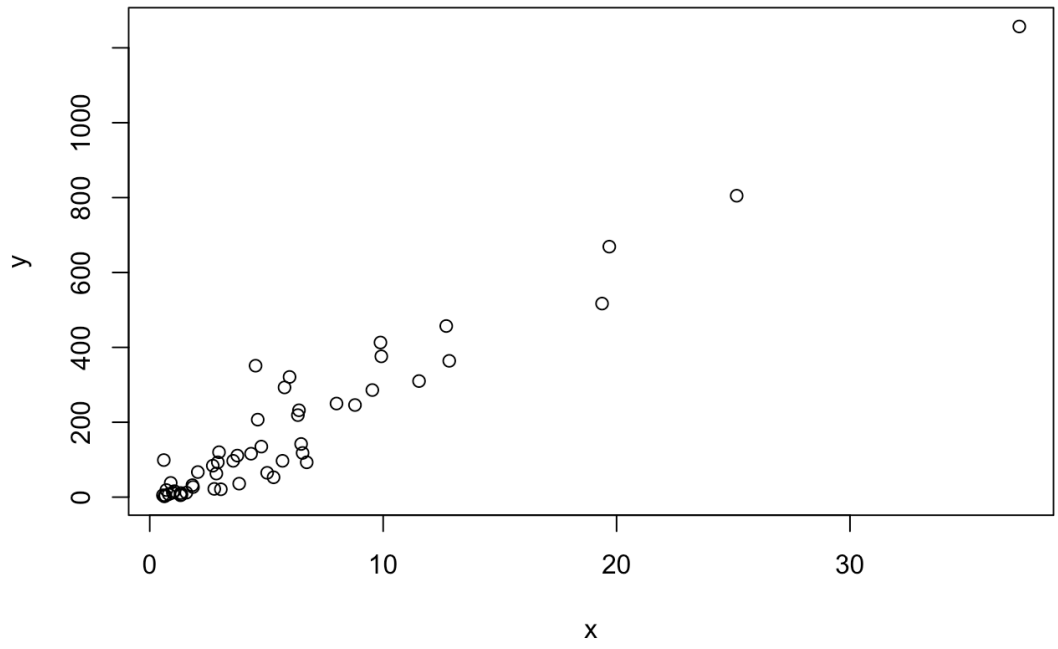
2.15 Basic plots

In Chapter 7 we describe an add-on package that provides a powerful approach to producing plots in R. We then have an entire part on Data Visualization in which we provide many examples. Here we briefly describe some of the functions that are available in a basic R installation.

2.15.1 plot

The plot function can be used to make scatterplots. Here is a plot of total murders versus population.

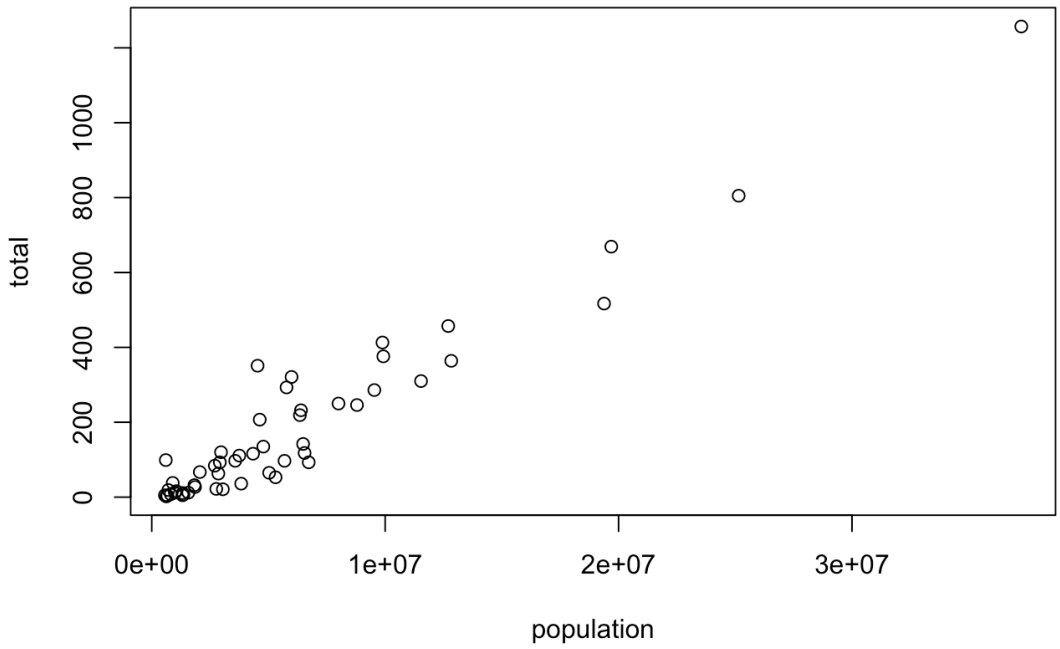
```
x <- murders$population / 10^6
y <- murders$total
plot(x, y)
```




```
# with(murders, plot(population/10^6, total)) is same.
```

For a quick plot that avoids accessing variables twice, we can use the with function:

```
with(murders, plot(population, total))
```

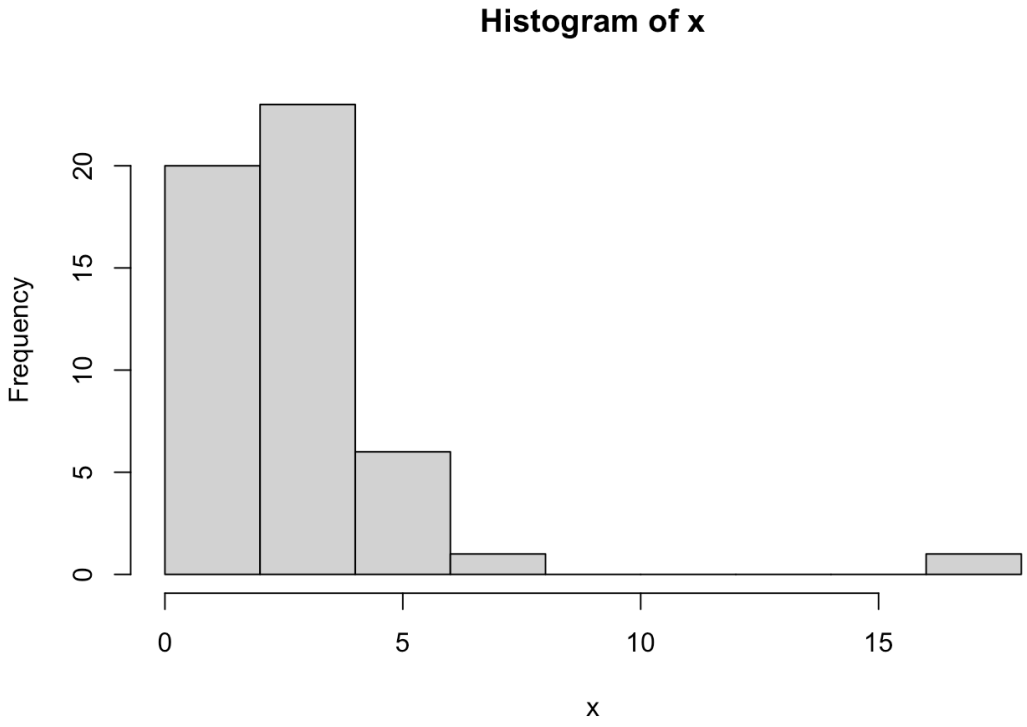


The function with lets us use the murders column names in the plot function. It also works with any data frames and any function.

2.15.2 hist

We will describe histograms as they relate to distributions in the Data Visualization part of the book. Here we will simply note that histograms are a powerful graphical summary of a list of numbers that gives you a general overview of the types of values you have. We can make a histogram of our murder rates by simply typing:

```
x <- with(murders, total / population * 100000)
hist(x)
```



We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

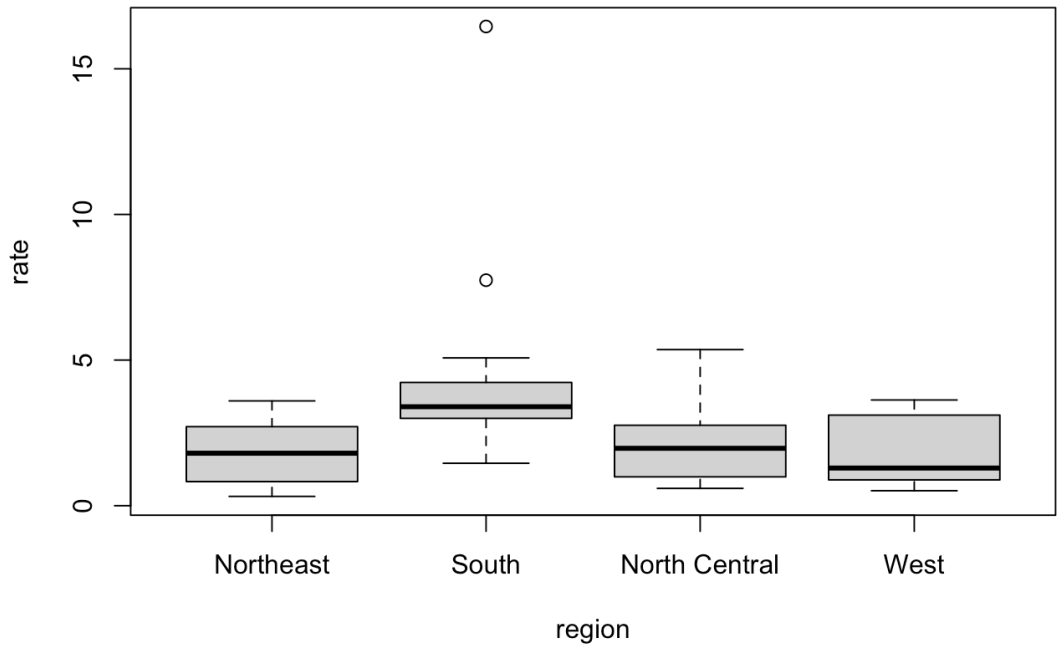
```
murders$state[which.max(x)]
```

```
## [1] "District of Columbia"
```

2.15.3 boxplot

Boxplots will also be described in the Data Visualization part of the book. They provide a more terse summary than histograms, but they are easier to stack with other boxplots. For example, here we can use them to compare the different regions:

```
murders$rate <- with(murders, total / population *
100000)
boxplot(rate~region, data = murders)
```

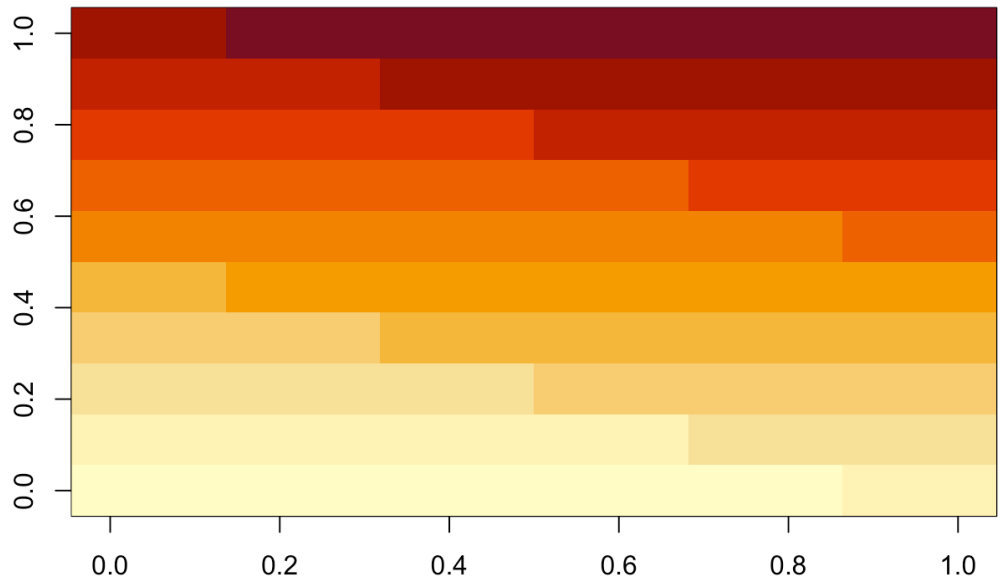


We can see that the South has higher murder rates than the other three regions.

2.15.4 image

The image function displays the values in a matrix using color. Here is a quick example:

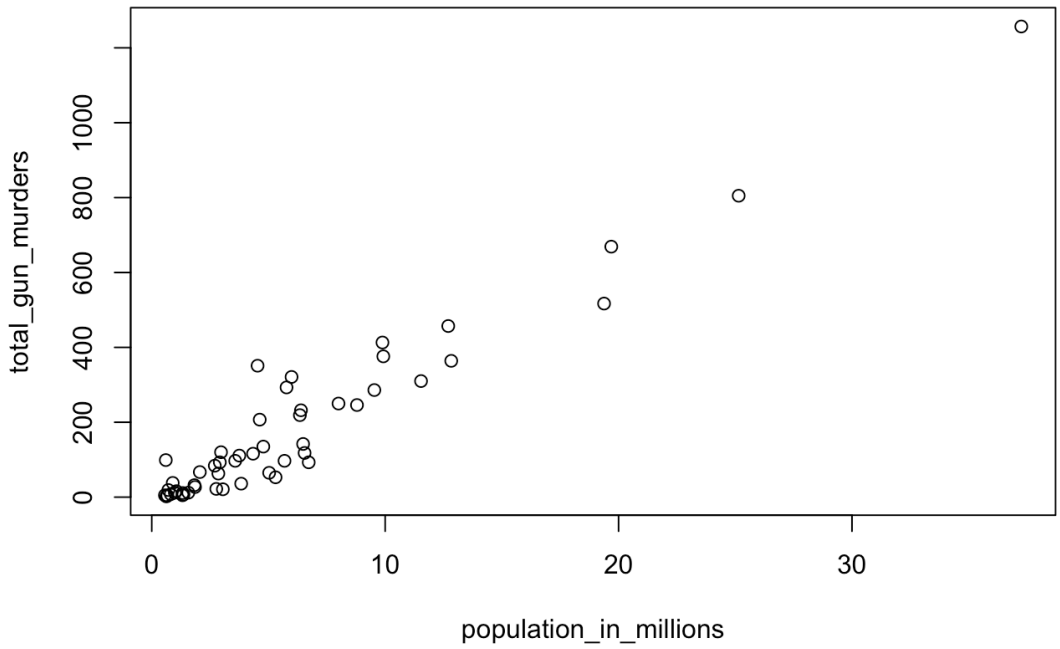
```
x <- matrix(1:120, 12, 10)
image(x)
```



2.16 Exercises

1. We made a plot of total murders versus population and noted a strong relationship. Not surprisingly, states with larger populations had more murders.

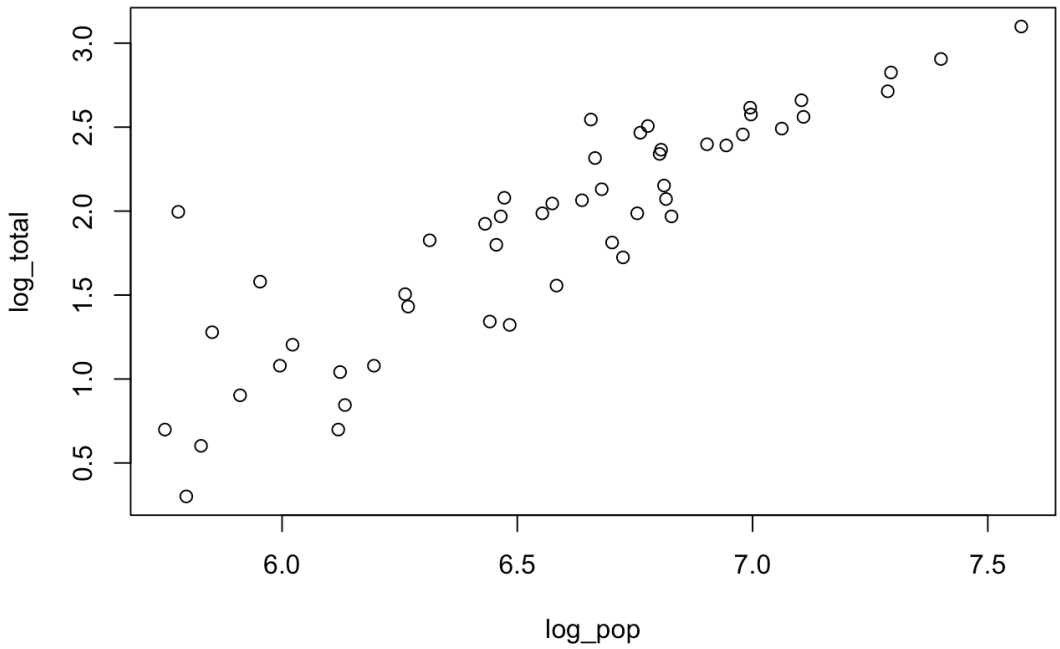
```
library(dslabs)
data(murders)
population_in_millions <- murders$population/10^6
total_gun_murders <- murders$total
plot(population_in_millions, total_gun_murders)
```



Keep in mind that many states have populations below 5 million and are bunched up. We may gain further insights from making this plot in the log scale. Transform the variables using the log10 transformation and then plot them.

```
#number1 answer:
?"log"
#log10(): 10 base
#log2(): 2 base
#numbers-only these two
#log(): with no number-e base... log(2.718281828) #>
[1]
log10(murders$population)
##  [1] 6.679404 5.851400 6.805638 6.464775 7.571172
6.701499 6.553166 5.953244
##  [9] 5.779397 7.294194 6.996512 6.133635 6.195230
7.108248 6.811830 6.483781
## [17] 6.455320 6.637426 6.656421 6.123316 6.761443
6.816084 6.994917 6.724597
```

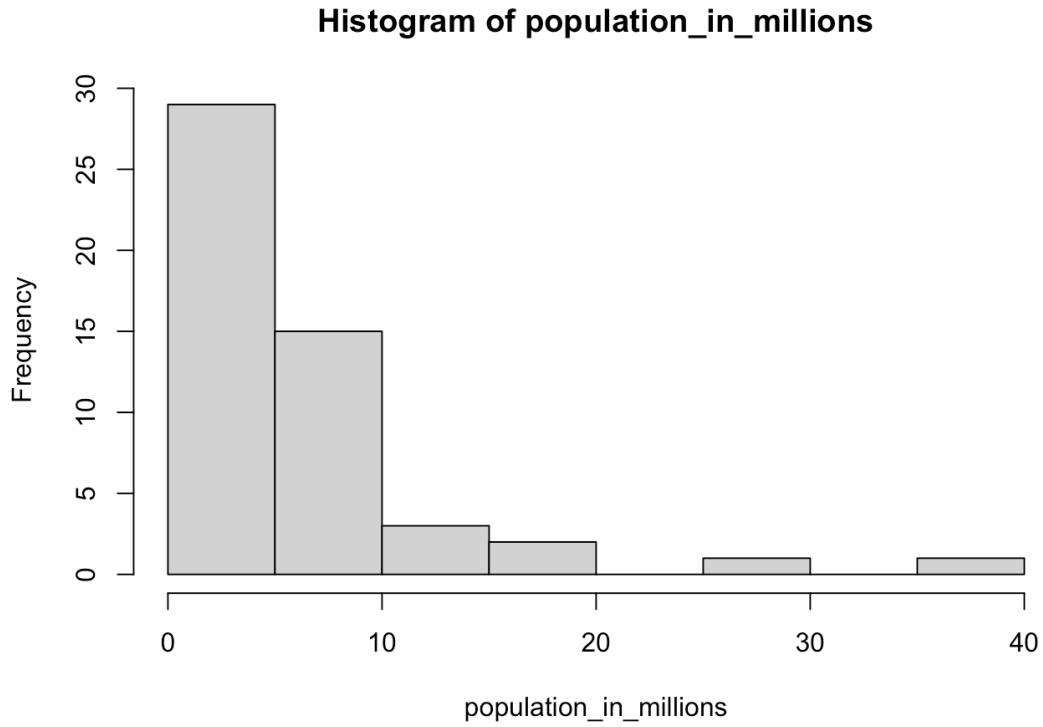
```
## [25] 6.472361 6.777349 5.995378 6.261582 6.431452
6.119411 6.944082 6.313694
## [33] 7.287311 6.979343 5.827751 7.062074 6.574188
6.583321 7.103885 6.022250
## [41] 6.665146 5.910720 6.802507 7.400461 6.441520
5.796395 6.903146 6.827663
## [49] 6.267874 6.754882 5.750991
log10(murders$total)
##  [1] 2.130334 1.278754 2.365488 1.968483 3.099335
1.812913 1.986772 1.579784
##  [9] 1.995635 2.825426 2.575188 0.845098 1.079181
2.561101 2.152288 1.322219
## [17] 1.799341 2.064458 2.545307 1.041393 2.466868
2.071882 2.615950 1.724276
## [25] 2.079181 2.506505 1.079181 1.505150 1.924279
0.698970 2.390935 1.826075
## [33] 2.713491 2.456366 0.602060 2.491362 2.045323
1.556303 2.659916 1.204120
## [41] 2.315970 0.903090 2.340444 2.905796 1.342423
0.301030 2.397940 1.968483
## [49] 1.431364 1.986772 0.698970
log_pop <- log10(murders$population)
log_total <- log10(murders$total)
plot(log_pop, log_total)
```



2. Create a histogram of the state populations.

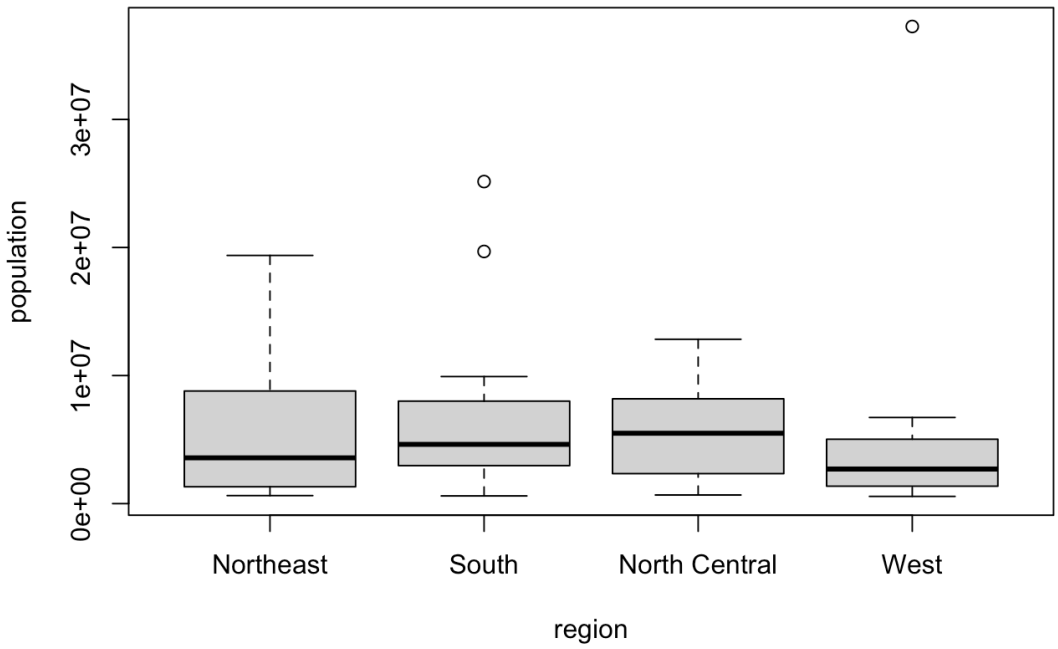
```
#number2 answer:
#case 1
hist(murders$population)
```


```
#case 2
population_in_millions <- murders$population/10^6
hist(population_in_millions)
```

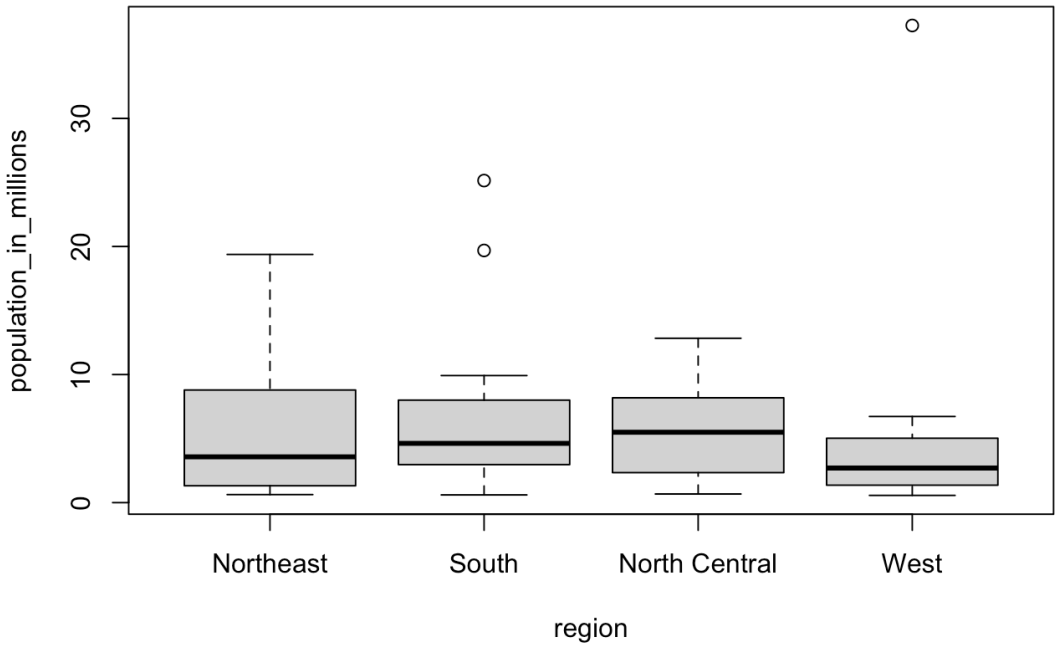


3. Generate boxplots of the state populations by region.

```
#number3 answer:
#case1
boxplot(population~region, data = murders)
```




```
#case2
boxplot(population_in_millions~region, data = murders)
```


population_in_millions
0
10
20
30
Northeast
South
North Central
West
region