Rafael A. Irizarry

Introduction to Data Science

1 Getting started with R and RStudio

1.1 Why R?

R is not a programming language like C or Java. It was not created by software engineers for software development. Instead, it was developed by statisticians as an interactive environment for data analysis. You can read the full history in the paper A Brief History of S[1]. The interactivity is an indispensable feature in data science because, as you will soon learn, the ability to quickly explore data is a necessity for success in this field. However, like in other programming languages, you can save your work as scripts that can be easily executed at any moment. These scripts serve as a record of the analysis you performed, a key feature that facilitates reproducible work. If you are an expert programmer, you should not expect R to follow the conventions you are used to since you will be disappointed. If you are patient, you will come to appreciate the unequal power of R when it comes to data analysis and, specifically, data visualization.

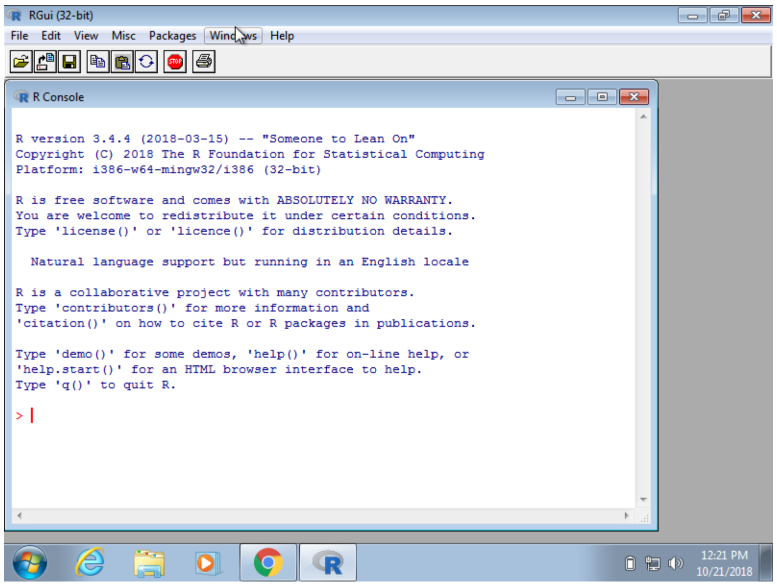
Other attractive features of R are:

1. R is free and open source[2].
2. It runs on all major platforms: Windows, Mac Os, UNIX/Linux.
3. Scripts and data objects can be shared seamlessly across platforms.
4. There is a large, growing, and active community of R users and, as a result, there are numerous resources for learning and asking questions[3] [4] [5].
5. It is easy for others to contribute add-ons which enables developers to share software implementations of new data science methodologies. This gives R users early access to the latest methods and to tools which are developed for a wide variety of disciplines, including ecology, molecular biology, social sciences, and geography, just to name a few examples.

1.2 The R console

Interactive data analysis usually occurs on the *R console* that executes commands as you type them. There are several ways to gain access to an R console. One way is to simply start R on your computer. The console looks something like this:

[1]https://pdfs.semanticscholar.org/9b48/46f192aa37ca122cfabb1ed1b59866d8bfda.pdf
[2]https://opensource.org/history
[3]https://stats.stackexchange.com/questions/138/free-resources-for-learning-r
[4]https://www.r-project.org/help.html
[5]https://stackoverflow.com/documentation/r/topics



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

As a quick example, try using the console to calculate a 15% tip on a meal that cost $19.71:

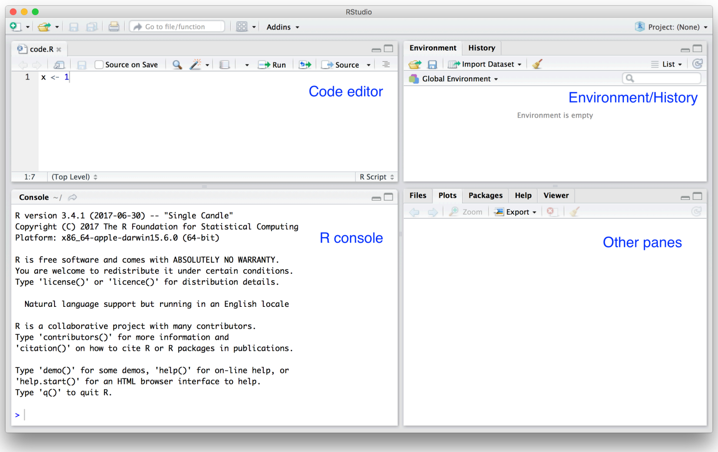
```
0.15 * 19.71
#> [1] 2.96
```

Note that in this book, grey boxes are used to show R code typed into the R console. The symbol #> is used to denote what the R console outputs.

1.3 Scripts

One of the great advantages of R over point-and-click analysis software is that you can save your work as scripts. You can edit and save these scripts using a text editor. The material in this book was developed using the interactive *integrated development environment* (IDE) RStudio[6]. RStudio includes an editor with many R specific features, a console to execute your code, and other useful panes, including one to show figures.

[6] https://www.rstudio.com/



```
x <- 1
```

```
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Most web-based R consoles also provide a pane to edit scripts, but not all permit you to save the scripts for later use.

All the R scripts used to generate this book can be found on GitHub[7].

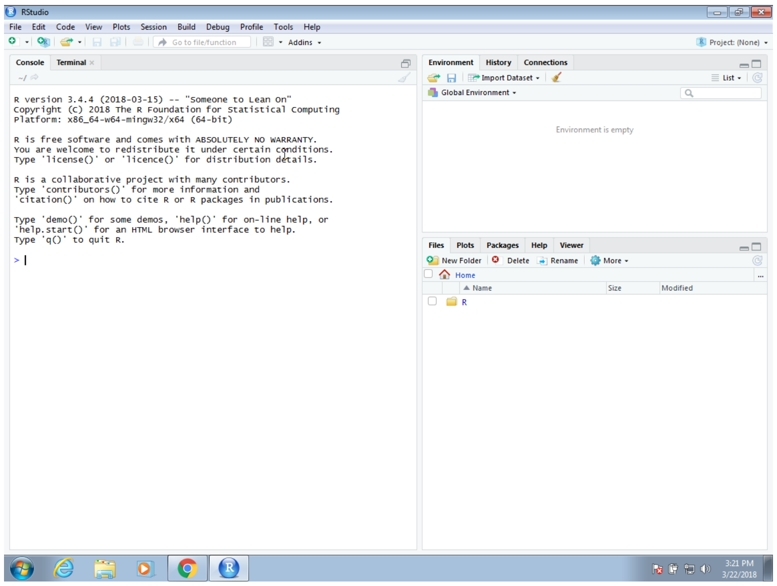
1.4 RStudio

RStudio will be our launching pad for data science projects. It not only provides an editor for us to create and edit our scripts but also provides many other useful tools. In this section, we go over some of the basics.

1.4.1 The panes

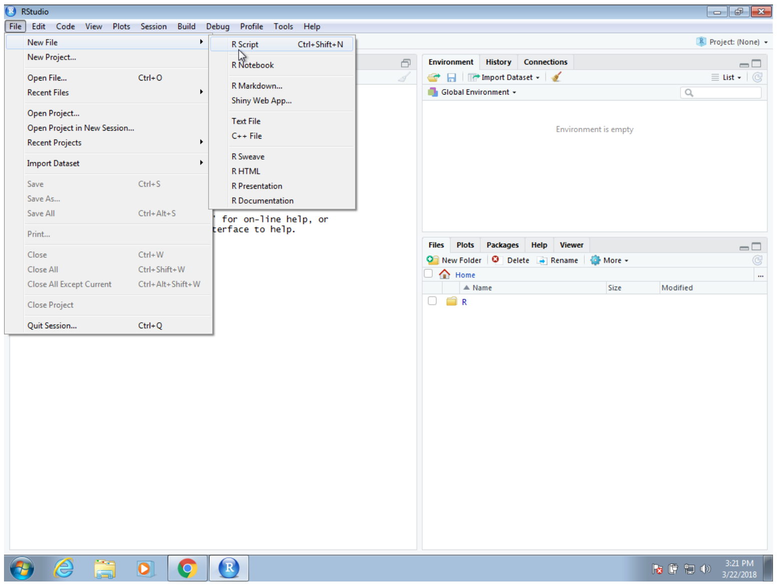
When you start RStudio for the first time, you will see three panes. The left pane shows the R console. On the right, the top pane includes tabs such as *Environment* and *History*, while the bottom pane shows five tabs: *File*, *Plots*, *Packages*, *Help*, and *Viewer* (these tabs may change in new versions). You can click on each tab to move across the different features.

[7]https://github.com/rafalab/dsbook

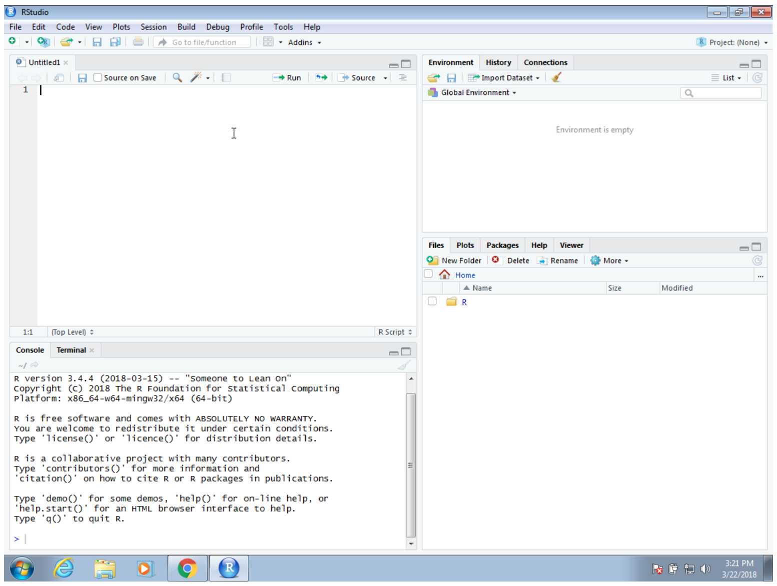


```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

To start a new script, you can click on File, the New File, then R Script.



This starts a new pane on the left and it is here where you can start writing your script.

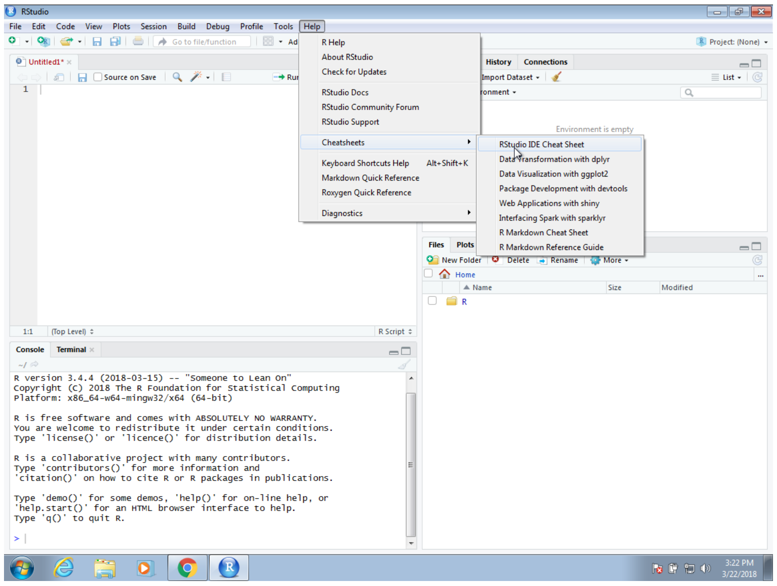


```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

1.4.2 Key bindings

Many tasks we perform with the mouse can be achieved with a combination of key strokes instead. These keyboard versions for performing tasks are referred to as *key bindings*. For example, we just showed how to use the mouse to start a new script, but you can also use a key binding: Ctrl+Shift+N on Windows and command+shift+N on the Mac.

Although in this tutorial we often show how to use the mouse, **we highly recommend that you memorize key bindings for the operations you use most**. RStudio provides a useful cheat sheet with the most widely used commands. You can get it from RStudio directly:



```
R version 3.4.4 (2018-03-15) -- "Someone to Lean On"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

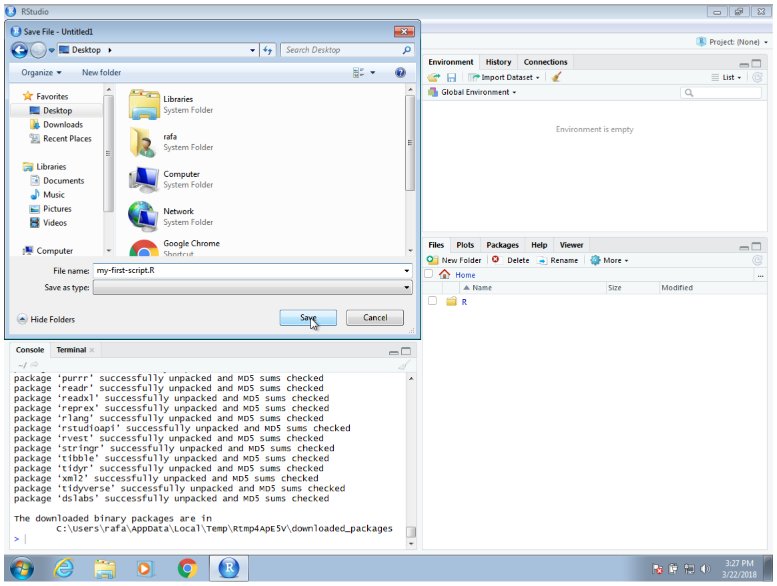
You might want to keep this handy so you can look up key-bindings when you find yourself performing repetitive point-and-clicking.

1.4.3 Running commands while editing scripts

There are many editors specifically made for coding. These are useful because color and indentation are automatically added to make code more readable. RStudio is one of these editors, and it was specifically developed for R. One of the main advantages provided by RStudio over other editors is that we can test our code easily as we edit our scripts. Below we show an example.

Let's start by opening a new script as we did before. A next step is to give the script a name. We can do this through the editor by saving the current new unnamed script. To do this, click on the save icon or use the key binding Ctrl+S on Windows and command+S on the Mac.

When you ask for the document to be saved for the first time, RStudio will prompt you for a name. A good convention is to use a descriptive name, with lower case letters, no spaces, only hyphens to separate words, and then followed by the suffix *.R*. We will call this script *my-first-script.R*.

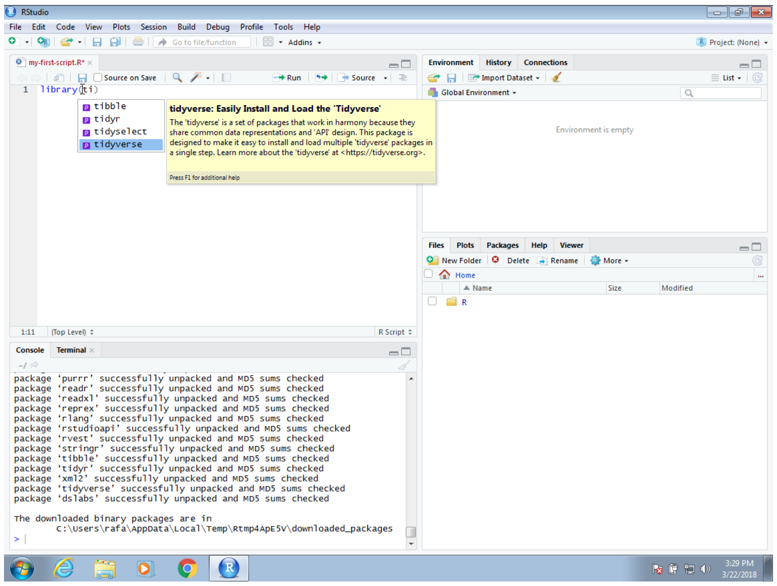


File name: my-first-script.R

Save as type:

```
package 'purrr' successfully unpacked and MD5 sums checked
package 'readr' successfully unpacked and MD5 sums checked
package 'readxl' successfully unpacked and MD5 sums checked
package 'reprex' successfully unpacked and MD5 sums checked
package 'rlang' successfully unpacked and MD5 sums checked
package 'rstudioapi' successfully unpacked and MD5 sums checked
package 'rvest' successfully unpacked and MD5 sums checked
package 'stringr' successfully unpacked and MD5 sums checked
package 'tibble' successfully unpacked and MD5 sums checked
package 'tidyr' successfully unpacked and MD5 sums checked
package 'xml2' successfully unpacked and MD5 sums checked
package 'tidyverse' successfully unpacked and MD5 sums checked
package 'dslabs' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\rafa\AppData\Local\Temp\Rtmp4ApE5V\downloaded_packages
> 
```

Now we are ready to start editing our first script. The first lines of code in an R script are dedicated to loading the libraries we will use. Another useful RStudio feature is that once we type `library()` it starts auto-completing with libraries that we have installed. Note what happens when we type `library(ti)`:



```
library(ti)
```

tidyverse: Easily Install and Load the 'Tidyverse'

The 'tidyverse' is a set of packages that work in harmony because they share common data representations and 'API' design. This package is designed to make it easy to install and load multiple 'tidyverse' packages in a single step. Learn more about the 'tidyverse' at <https://tidyverse.org>.

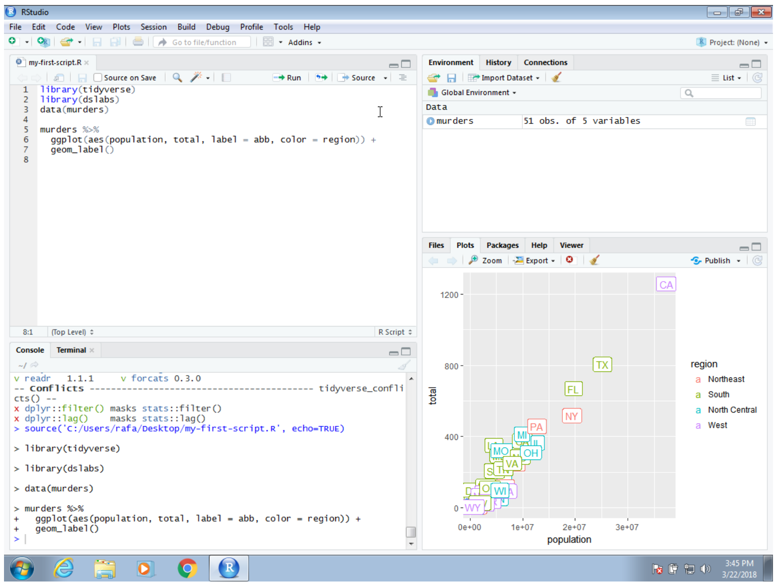
Press F1 for additional help

```
package ‘purrr’ successfully unpacked and MD5 sums checked
package ‘readr’ successfully unpacked and MD5 sums checked
package ‘readxl’ successfully unpacked and MD5 sums checked
package ‘reprex’ successfully unpacked and MD5 sums checked
package ‘rlang’ successfully unpacked and MD5 sums checked
package ‘rstudioapi’ successfully unpacked and MD5 sums checked
package ‘rvest’ successfully unpacked and MD5 sums checked
package ‘stringr’ successfully unpacked and MD5 sums checked
package ‘tibble’ successfully unpacked and MD5 sums checked
package ‘tidyr’ successfully unpacked and MD5 sums checked
package ‘xml2’ successfully unpacked and MD5 sums checked
package ‘tidyverse’ successfully unpacked and MD5 sums checked
package ‘dslabs’ successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\rafa\AppData\Local\Temp\Rtmp4ApE5V\downloaded_packages
> 
```

Another feature you may have noticed is that when you type `library(` the second parenthesis is automatically added. This will help you avoid one of the most common errors in coding: forgetting to close a parenthesis.

Now we can continue to write code. As an example, we will make a graph showing murder totals versus population totals by state. Once you are done writing the code needed to make this plot, you can try it out by *executing* the code. To do this, click on the *Run* button on the upper right side of the editing pane. You can also use the key binding: Ctrl+Shift+Enter on Windows or command+shift+return on the Mac.

Once you run the code, you will see it appear in the R console and, in this case, the generated plot appears in the plots console. Note that the plot console has a useful interface that permits you to click back and forward across different plots, zoom in to the plot, or save the plots as files.



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

murders %>%
  ggplot(aes(population, total, label = abb, color = region)) +
  geom_label()
```

```
murders    51 obs. of 5 variables
```

```
v readr   1.1.1     v forcats 0.3.0
-- Conflicts ------------------------------------ tidyverse_confli
cts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
> source('C:/Users/rafa/Desktop/my-first-script.R', echo=TRUE)

> library(tidyverse)

> library(dslabs)

> data(murders)

> murders %>%
+   ggplot(aes(population, total, label = abb, color = region)) +
+   geom_label()
> 
```

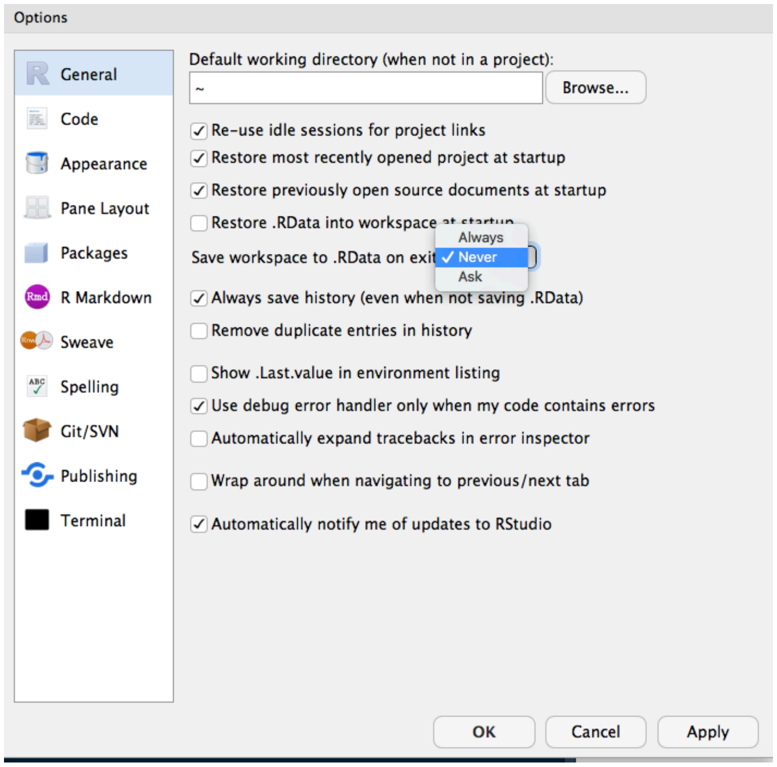
To run one line at a time instead of the entire script, you can use Control-Enter on Windows and command-return on the Mac.

1.4.4 Changing global options

You can change the look and functionality of RStudio quite a bit.

To change the global options you click on *Tools* then *Global Options…*.

As an example we show how to make a change that we **highly recommend**. This is to change the *Save workspace to .RData on exit* to *Never* and uncheck the *Restore .RData into workspace at start*. By default, when you exit R saves all the objects you have created into a file called .RData. This is done so that when you restart the session in the same folder, it will load these objects. We find that this causes confusion especially when we share code with colleagues and assume they have this .RData file. To change these options, make your *General* settings look like this:



Default working directory (when not in a project): ~

✓ Re-use idle sessions for project links
✓ Restore most recently opened project at startup
✓ Restore previously open source documents at startup
☐ Restore .RData into workspace at startup
Save workspace to .RData on exit: Always [✓ Never] Ask
✓ Always save history (even when not saving .RData)
☐ Remove duplicate entries in history
☐ Show .Last.value in environment listing
✓ Use debug error handler only when my code contains errors
☐ Automatically expand tracebacks in error inspector
☐ Wrap around when navigating to previous/next tab
✓ Automatically notify me of updates to RStudio

1.5 Installing R packages

The functionality provided by a fresh install of R is only a small fraction of what is possible. In fact, we refer to what you get after your first install as *base R*. The extra functionality comes from add-ons available from developers. There are currently hundreds of these available from CRAN and many others shared via other repositories such as GitHub. However, because not everybody needs all available functionality, R instead makes different components available via *packages*. R makes it very easy to install packages from within R. For example, to install the **dslabs** package, which we use to share datasets and code related to this book, you would type:

```
install.packages("dslabs")
```

In RStudio, you can navigate to the *Tools* tab and select install packages. We can then load the package into our R sessions using the `library` function:

```
library(dslabs)
```

As you go through this book, you will see that we load packages without installing them. This is because once you install a package, it remains installed and only needs to be loaded with `library`. The package remains loaded until we quit the R session. If you try to load a package and get an error, it probably means you need to install it first.

We can install more than one package at once by feeding a character vector to this function:

```
install.packages(c("tidyverse", "dslabs"))
```

Note that installing **tidyverse** actually installs several packages. This commonly occurs when a package has *dependencies*, or uses functions from other packages. When you load a package using `library`, you also load its dependencies.

Once packages are installed, you can load them into R and you do not need to install them again, unless you install a fresh version of R. Remember packages are installed in R not RStudio.

It is helpful to keep a list of all the packages you need for your work in a script because if you need to perform a fresh install of R, you can re-install all your packages by simply running a script.

You can see all the packages you have installed using the following function:

```
installed.packages()
```

Part I

R

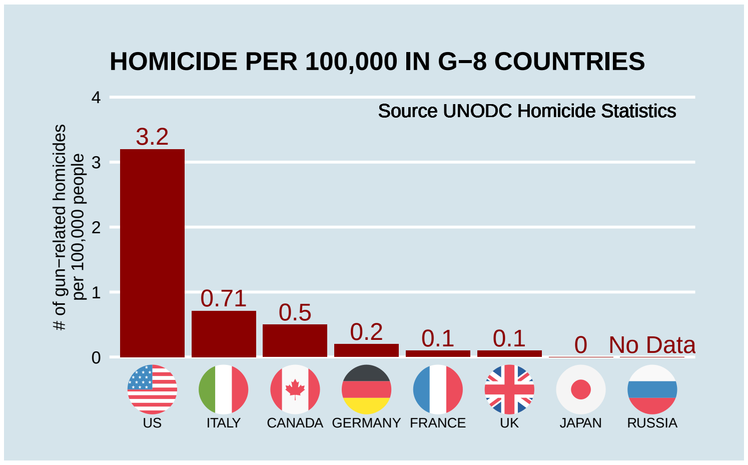
2

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[1]. 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[2]. Both R and RStudio are free and available online.

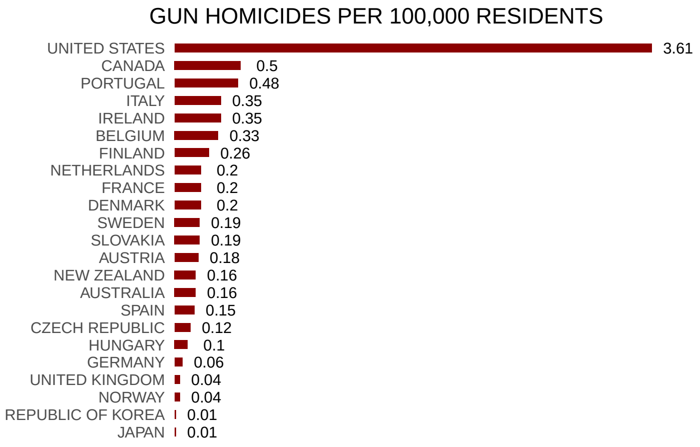
2.1 Case study: US Gun Murders

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**[3] have you worried. Charts like this may concern you even more:

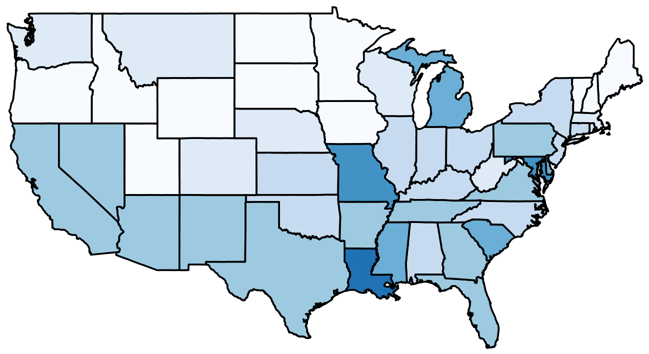


Or even worse, this version from everytown.org:

[1]https://rstudio.cloud
[2]https://rafalab.github.io/dsbook/installing-r-rstudio.html
[3]http://abcnews.go.com/blogs/headlines/2012/12/us-gun-ownership-homicide-rate-higher-than-other-developed-countries/



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:

$$\frac{-b - \sqrt{b^2 - 4ac}}{2a} \text{ and } \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

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
```

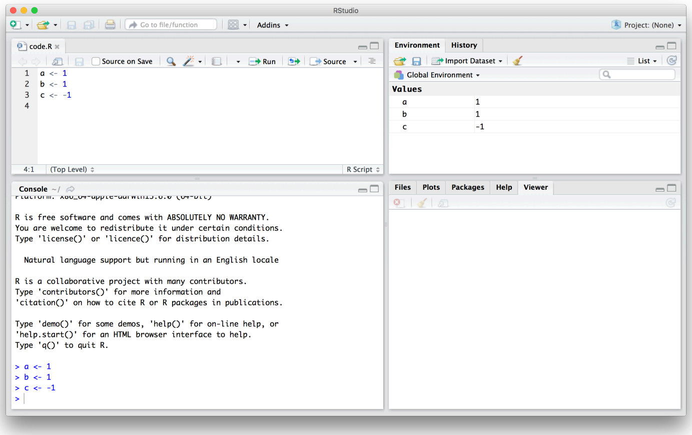
We use the term *object* to describe stuff that is stored in R. Variables are examples, but objects can also be more complicated entities such as functions, which are described later.

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"          "dat"        "img_path" "murders"
```

In RStudio, the *Environment* tab shows the values:



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

```
Values
a        1
b        1
c        -1
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

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

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`.

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] 0.618
(-b - sqrt(b^2 - 4*a*c) ) / ( 2*a )
#> [1] -1.62
```

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] 2.08
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
```

R will show you Mauna Loa atmospheric CO2 concentration data.

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

```
pi
#> [1] 3.14
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[4].

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

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.

[4] http://adv-r.had.co.nz/Style.html

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

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?

2. Now use the same formula to compute the sum of the integers from 1 through 1,000.

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

```
n <- 1000
x <- seq(1, n)
sum(x)
```

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

a. `sum` creates a list of numbers and `seq` adds them up.
b. `seq` creates a list of numbers and `sum` adds them up.
c. `seq` creates a random list and `sum` computes the sum of 1 through 1,000.
d. `sum` always returns the same number.

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.

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.

a. `log(10^x)`
b. `log10(x^10)`
c. `log(exp(x))`
d. `exp(log(x, base = 2))`

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)
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

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
#>  [8]   897934   601723 19687653  9920000  1360301  1567582 12830632
#> [15]  6483802  3046355  2853118  4339367  4533372  1328361  5773552
#> [22]  6547629  9883640  5303925  2967297  5988927   989415  1826341
#> [29]  2700551  1316470  8791894  2059179 19378102  9535483   672591
#> [36] 11536504  3751351  3831074 12702379  1052567  4625364   814180
#> [43]  6346105 25145561  2763885   625741  8001024  6724540  1852994
#> [50]  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.

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)`

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 is for the levels to follow alphabetical order. However, often we want the levels to follow a different order. We will see several examples of this in the Data Visualization part of the book. 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.

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*. We will cover lists in more detail later, but know that they are useful because you can store any combination of different types. Below is an example of a list we created for you:

```
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 `$`. In fact, data frames are a type of list.

```
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
#> [[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)
#>   V1 V2 V3
#> 1  1  5  9
#> 2  2  6 10
#> 3  3  7 11
#> 4  4  8 12
```

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, ]
#>     state abb region population total
#> 2  Alaska  AK   West     710231    19
#> 3 Arizona  AZ   West    6392017   232
```

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?

a. The 51 states.
b. The murder rates for all 50 states and DC.
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.
d. `str` shows no relevant information.

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

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

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.

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

```
class(murders$region)
```

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

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.

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
#>  italy canada  egypt
#>    380    124    818
```

The object `codes` continues to be a numeric vector:

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

but with names:

```
names(codes)
#> [1] "italy"  "canada" "egypt"
```

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
#>  italy canada  egypt
#>    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
#>  italy canada  egypt
#>    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]
#> canada
#>    124
```

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

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

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

```
codes[1:2]
#>  italy canada
#>    380    124
```

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

```
codes["canada"]
#> canada
#>    124
codes[c("egypt","italy")]
#> egypt italy
#>   818   380
```

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 `NA`s 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`.

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

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

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

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

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

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

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`.

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

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

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.

12. Define the following vector:

```
x <- c("1", "3", "5")
```

and coerce it to get integers.

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
#> [14]   27   32   36   38   53   63   65   67   84   93   93   97   97
#> [27]   99  111  116  118  120  135  142  207  219  232  246  250  286
#> [40]  293  310  321  351  364  376  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"
#> [14] "WV" "NE" "OR" "DE" "MN" "KS" "CO" "NM" "NV" "AR" "WA" "CT" "WI"
#> [27] "DC" "OK" "KY" "MA" "MS" "AL" "IN" "SC" "TN" "AZ" "NJ" "VA" "NC"
#> [40] "MD" "OH" "MO" "LA" "IL" "GA" "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:

original	sort	order	rank
31	4	2	3
4	15	3	1
15	31	1	2
92	65	5	5
65	92	4	4

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.

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`.

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.

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.

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`.

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.

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 `NA`s does `na_example` have.

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

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] 175 157 168 178 178 185 170 185 170 178
```

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:

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} + \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a+e \\ b+f \\ c+g \\ d+h \end{pmatrix}$$

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"
#> [14] "CO" "WA" "WV" "RI" "WI" "NE" "MA" "IN" "KS" "NY" "KY" "AK" "OH"
#> [27] "CT" "NJ" "AL" "IL" "OK" "NC" "NV" "VA" "AR" "TX" "NM" "CA" "FL"
#> [40] "TN" "PA" "AZ" "GA" "MS" "MI" "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 $C = \frac{5}{9} \times (F - 32)$.

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

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?

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 & 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] 3.37
```

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] 2.67 3.40 3.20
```

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
```

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.

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.

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

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 `&`.

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?

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.

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?

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.

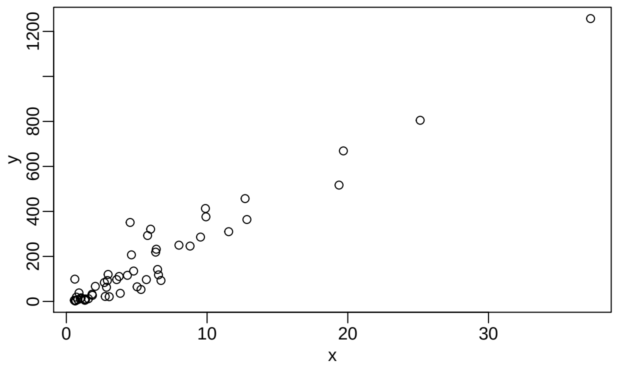
2.15 Basic plots

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)
```



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

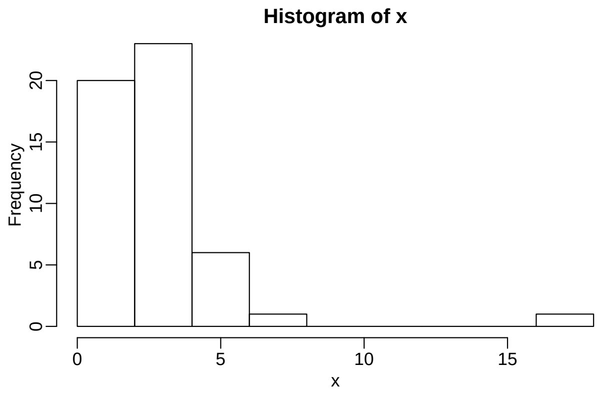
```
with(murders, plot(population, total))
```

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)
```



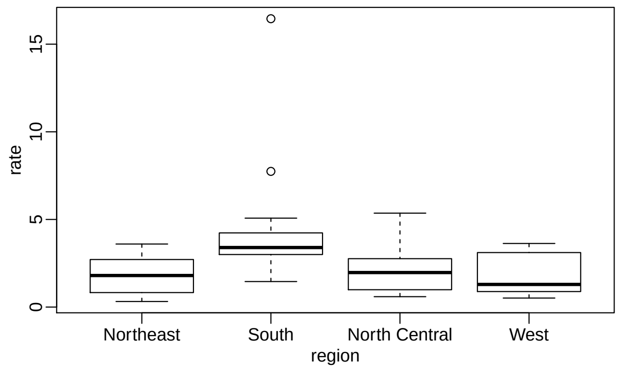
We can see that there is a wide range of values with most of them between 2 and 3 and one very extreme case with a murder rate of more than 15:

```
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)
```

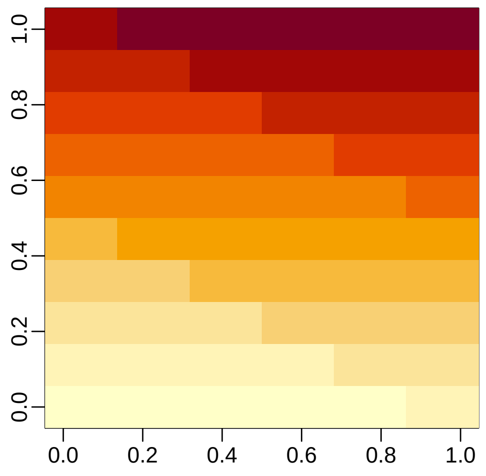


We can see that the South has higher murder rates than the other three regions.

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.

2. Create a histogram of the state populations.

3. Generate boxplots of the state populations by region.

3

Programming basics

We teach R because it greatly facilitates data analysis, the main topic of this book. By coding in R, we can efficiently perform exploratory data analysis, build data analysis pipelines, and prepare data visualization to communicate results. However, R is not just a data analysis environment but a programming language. Advanced R programmers can develop complex packages and even improve R itself, but we do not cover advanced programming in this book. Nonetheless, in this section, we introduce three key programming concepts: conditional expressions, for-loops, and functions. These are not just key building blocks for advanced programming, but are sometimes useful during data analysis. We also note that there are several functions that are widely used to program in R but that we will not cover in this book. These include `split`, `cut`, `do.call`, and `Reduce`, as well as the **data.table** package. These are worth learning if you plan to become an expert R programmer.

3.1 Conditional expressions

Conditional expressions are one of the basic features of programming. They are used for what is called *flow control*. The most common conditional expression is the if-else statement. In R, we can actually perform quite a bit of data analysis without conditionals. However, they do come up occasionally, and you will need them once you start writing your own functions and packages.

Here is a very simple example showing the general structure of an if-else statement. The basic idea is to print the reciprocal of `a` unless `a` is 0:

```
a <- 0

if(a!=0){
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
#> [1] "No reciprocal for 0."
```

Let's look at one more example using the US murders data frame:

```
library(dslabs)
data(murders)
murder_rate <- murders$total / murders$population*100000
```

Here is a very simple example that tells us which states, if any, have a murder rate lower than 0.5 per 100,000. The `if` statement protects us from the case in which no state satisfies the condition.

```
ind <- which.min(murder_rate)

if(murder_rate[ind] < 0.5){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
#> [1] "Vermont"
```

If we try it again with a rate of 0.25, we get a different answer:

```
if(murder_rate[ind] < 0.25){
  print(murders$state[ind])
} else{
  print("No state has a murder rate that low.")
}
#> [1] "No state has a murder rate that low."
```

A related function that is very useful is `ifelse`. This function takes three arguments: a logical and two possible answers. If the logical is `TRUE`, the value in the second argument is returned and if `FALSE`, the value in the third argument is returned. Here is an example:

```
a <- 0
ifelse(a > 0, 1/a, NA)
#> [1] NA
```

The function is particularly useful because it works on vectors. It examines each entry of the logical vector and returns elements from the vector provided in the second argument, if the entry is `TRUE`, or elements from the vector provided in the third argument, if the entry is `FALSE`.

```
a <- c(0, 1, 2, -4, 5)
result <- ifelse(a > 0, 1/a, NA)
```

This table helps us see what happened:

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	-0.25	NA	NA
5	TRUE	0.20	NA	0.2

Here is an example of how this function can be readily used to replace all the missing values in a vector with zeros:

```
data(na_example)
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
#> [1] 0
```

Two other useful functions are `any` and `all`. The `any` function takes a vector of logicals and returns `TRUE` if any of the entries is `TRUE`. The `all` function takes a vector of logicals and returns `TRUE` if all of the entries are `TRUE`. Here is an example:

```
z <- c(TRUE, TRUE, FALSE)
any(z)
#> [1] TRUE
all(z)
#> [1] FALSE
```

3.2 Defining functions

As you become more experienced, you will find yourself needing to perform the same operations over and over. A simple example is computing averages. We can compute the average of a vector `x` using the `sum` and `length` functions: `sum(x)/length(x)`. Because we do this repeatedly, it is much more efficient to write a function that performs this operation. This particular operation is so common that someone already wrote the `mean` function and it is included in base R. However, you will encounter situations in which the function does not already exist, so R permits you to write your own. A simple version of a function that computes the average can be defined like this:

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

Now `avg` is a function that computes the mean:

```
x <- 1:100
identical(mean(x), avg(x))
#> [1] TRUE
```

Notice that variables defined inside a function are not saved in the workspace. So while we use `s` and `n` when we call `avg`, the values are created and changed only during the call. Here is an illustrative example:

```
s <- 3
avg(1:10)
#> [1] 5.5
s
#> [1] 3
```

Note how `s` is still 3 after we call `avg`.

In general, functions are objects, so we assign them to variable names with `<-`. The function `function` tells R you are about to define a function. The general form of a function definition looks like this:

```
my_function <- function(VARIABLE_NAME){
  perform operations on VARIABLE_NAME and calculate VALUE
  VALUE
}
```

The functions you define can have multiple arguments as well as default values. For example, we can define a function that computes either the arithmetic or geometric average depending on a user defined variable like this:

```
avg <- function(x, arithmetic = TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
```

We will learn more about how to create functions through experience as we face more complex tasks.

3.3 Namespaces

Once you start becoming more of an R expert user, you will likely need to load several add-on packages for some of your analysis. Once you start doing this, it is likely that two packages use the same name for two different functions. And often these functions do completely different things. In fact, you have already encountered this because both **dplyr** and the R-base **stats** package define a `filter` function. There are five other examples in **dplyr**. We know this because when we first load **dplyr** we see the following message:

```
The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```

So what does R do when we type `filter`? Does it use the **dplyr** function or the **stats** function? From our previous work we know it uses the **dplyr** one. But what if we want to use the **stats** version?

These functions live in different *namespaces*. R will follow a certain order when searching for a function in these *namespaces*. You can see the order by typing:

```
search()
```

The first entry in this list is the global environment which includes all the objects you define.

So what if we want to use the **stats** `filter` instead of the **dplyr** filter but **dplyr** appears first in the search list? You can force the use of a specific name space by using double colons (`::`) like this:

```
stats::filter
```

If we want to be absolutely sure we use the **dplyr** `filter` we can use

```
dplyr::filter
```

Also note that if we want to use a function in a package without loading the entire package, we can use the double colon as well.

For more on this more advanced topic we recommend the R packages book[1].

3.4 For-loops

The formula for the sum of the series $1 + 2 + \cdots + n$ is $n(n+1)/2$. What if we weren't sure that was the right function? How could we check? Using what we learned about functions we can create one that computes the S_n:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
```

How can we compute S_n for various values of n, say $n = 1, \dots, 25$? Do we write 25 lines of code calling `compute_s_n`? No, that is what for-loops are for in programming. In this case, we are performing exactly the same task over and over, and the only thing that is changing is the value of n. For-loops let us define the range that our variable takes (in our example $n = 1, \dots, 10$), then change the value and evaluate expression as you *loop*.

Perhaps the simplest example of a for-loop is this useless piece of code:

```
for(i in 1:5){
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

[1] http://r-pkgs.had.co.nz/namespace.html

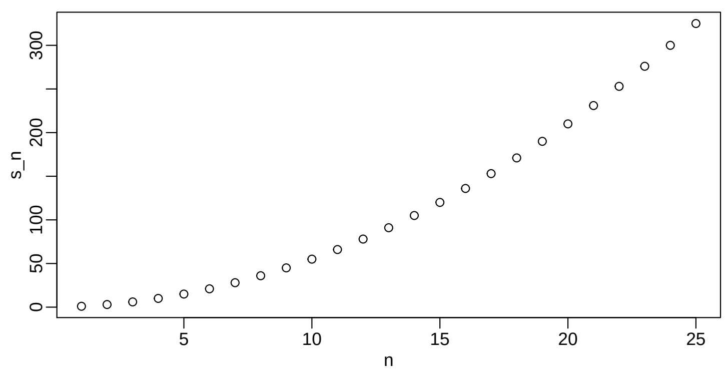
Here is the for-loop we would write for our S_n example:

```
m <- 25
s_n <- vector(length = m) # create an empty vector
for(n in 1:m){
  s_n[n] <- compute_s_n(n)
}
```

In each iteration $n = 1$, $n = 2$, etc…, we compute S_n and store it in the nth entry of `s_n`.

Now we can create a plot to search for a pattern:

```
n <- 1:m
plot(n, s_n)
```



If you noticed that it appears to be a quadratic, you are on the right track because the formula is $n(n+1)/2$.

3.5 Vectorization and functionals

Although for-loops are an important concept to understand, in R we rarely use them. As you learn more R, you will realize that *vectorization* is preferred over for-loops since it results in shorter and clearer code. We already saw examples in the Vector Arithmetic section. A *vectorized* function is a function that will apply the same operation on each of the vectors.

```
x <- 1:10
sqrt(x)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
y <- 1:10
x*y
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

To make this calculation, there is no need for for-loops. However, not all functions work this way. For instance, the function we just wrote, `compute_s_n`, does not work element-wise since it is expecting a scalar. This piece of code does not run the function on each entry of `n`:

```
n <- 1:25
compute_s_n(n)
```

Functionals are functions that help us apply the same function to each entry in a vector, matrix, data frame, or list. Here we cover the functional that operates on numeric, logical, and character vectors: `sapply`.

The function `sapply` permits us to perform element-wise operations on any function. Here is how it works:

```
x <- 1:10
sapply(x, sqrt)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

Each element of `x` is passed on to the function `sqrt` and the result is returned. These results are concatenated. In this case, the result is a vector of the same length as the original `x`. This implies that the for-loop above can be written as follows:

```
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

Other functionals are `apply`, `lapply`, `tapply`, `mapply`, `vapply`, and `replicate`. We mostly use `sapply`, `apply`, and `replicate` in this book, but we recommend familiarizing yourselves with the others as they can be very useful.

3.6 Exercises

1. What will this conditional expression return?

```
x <- c(1,2,-3,4)

if(all(x>0)){
  print("All Postives")
} else{
  print("Not all positives")
}
```

2. Which of the following expressions is always `FALSE` when at least one entry of a logical vector `x` is TRUE?

a. `all(x)`
b. `any(x)`

c. `any(!x)`
d. `all(!x)`

3. The function `nchar` tells you how many characters long a character vector is. Write a line of code that assigns to the object `new_names` the state abbreviation when the state name is longer than 8 characters.

4. Create a function `sum_n` that for any given value, say n, computes the sum of the integers from 1 to n (inclusive). Use the function to determine the sum of integers from 1 to 5,000.

5. Create a function `altman_plot` that takes two arguments, `x` and `y`, and plots the difference against the sum.

6. After running the code below, what is the value of `x`?

```
x <- 3
my_func <- function(y){
  x <- 5
  y+5
}
```

7. Write a function `compute_s_n` that for any given n computes the sum $S_n = 1^2 + 2^2 + 3^2 + \dots n^2$. Report the value of the sum when $n = 10$.

8. Define an empty numerical vector `s_n` of size 25 using `s_n <- vector("numeric", 25)` and store in the results of $S_1, S_2, \dots S_{25}$ using a for-loop.

9. Repeat exercise 8, but this time use `sapply`.

10. Repeat exercise 8, but this time use `map_dbl`.

11. Plot S_n versus n. Use points defined by $n = 1, \dots, 25$.

12. Confirm that the formula for this sum is $S_n = n(n+1)(2n+1)/6$.

4

The tidyverse

Up to now we have been manipulating vectors by reordering and subsetting them through indexing. However, once we start more advanced analyses, the preferred unit for data storage is not the vector but the data frame. In this chapter we learn to work directly with data frames, which greatly facilitate the organization of information. We will be using data frames for the majority of this book. We will focus on a specific data format referred to as *tidy* and on specific collection of packages that are particularly helpful for working with *tidy* data referred to as the *tidyverse*.

We can load all the tidyverse packages at once by installing and loading the **tidyverse** package:

```
library(tidyverse)
```

We will learn how to implement the tidyverse approach throughout the book, but before delving into the details, in this chapter we introduce some of the most widely used tidyverse functionality, starting with the **dplyr** package for manipulating data frames and the **purrr** package for working with functions. Note that the tidyverse also includes a graphing package, **ggplot2**, which we introduce later in Chapter 7 in the Data Visualization part of the book; the **readr** package discussed in Chapter 5; and many others. In this chapter, we first introduce the concept of *tidy data* and then demonstrate how we use the tidyverse to work with data frames in this format.

4.1 Tidy data

We say that a data table is in *tidy* format if each row represents one observation and columns represent the different variables available for each of these observations. The `murders` dataset is an example of a tidy data frame.

```
#>        state abb region population total
#> 1    Alabama  AL  South    4779736   135
#> 2     Alaska  AK   West     710231    19
#> 3    Arizona  AZ   West    6392017   232
#> 4   Arkansas  AR  South    2915918    93
#> 5 California  CA   West   37253956  1257
#> 6   Colorado  CO   West    5029196    65
```

Each row represent a state with each of the five columns providing a different variable related to these states: name, abbreviation, region, population, and total murders.

To see how the same information can be provided in different formats, consider the following example:

```
#>       country year fertility
#> 1     Germany 1960      2.41
#> 2 South Korea 1960      6.16
#> 3     Germany 1961      2.44
#> 4 South Korea 1961      5.99
#> 5     Germany 1962      2.47
#> 6 South Korea 1962      5.79
```

This tidy dataset provides fertility rates for two countries across the years. This is a tidy dataset because each row presents one observation with the three variables being country, year, and fertility rate. However, this dataset originally came in another format and was reshaped for the **dslabs** package. Originally, the data was in the following format:

```
#>       country 1960 1961 1962
#> 1     Germany 2.41 2.44 2.47
#> 2 South Korea 6.16 5.99 5.79
```

The same information is provided, but there are two important differences in the format: 1) each row includes several observations and 2) one of the variables, year, is stored in the header. For the tidyverse packages to be optimally used, data need to be reshaped into `tidy` format, which you will learn to do in the Data Wrangling part of the book. Until then, we will use example datasets that are already in tidy format.

Although not immediately obvious, as you go through the book you will start to appreciate the advantages of working in a framework in which functions use tidy formats for both inputs and outputs. You will see how this permits the data analyst to focus on more important aspects of the analysis rather than the format of the data.

4.2 Exercises

1. Examine the built-in dataset `co2`. Which of the following is true:

   a. `co2` is tidy data: it has one year for each row.
   b. `co2` is not tidy: we need at least one column with a character vector.
   c. `co2` is not tidy: it is a matrix instead of a data frame.
   d. `co2` is not tidy: to be tidy we would have to wrangle it to have three columns (year, month and value), then each co2 observation would have a row.

2. Examine the built-in dataset `ChickWeight`. Which of the following is true:

   a. `ChickWeight` is not tidy: each chick has more than one row.
   b. `ChickWeight` is tidy: each observation (a weight) is represented by one row. The chick from which this measurement came is one of the variables.

c. `ChickWeight` is not tidy: we are missing the year column.
d. `ChickWeight` is tidy: it is stored in a data frame.

3. Examine the built-in dataset `BOD`. Which of the following is true:

a. `BOD` is not tidy: it only has six rows.
b. `BOD` is not tidy: the first column is just an index.
c. `BOD` is tidy: each row is an observation with two values (time and demand)
d. `BOD` is tidy: all small datasets are tidy by definition.

4. Which of the following built-in datasets is tidy (you can pick more than one):

a. `BJsales`
b. `EuStockMarkets`
c. `DNase`
d. `Formaldehyde`
e. `Orange`
f. `UCBAdmissions`

4.3 Manipulating data frames

The **dplyr** package from the **tidyverse** introduces functions that perform some of the most common operations when working with data frames and uses names for these functions that are relatively easy to remember. For instance, to change the data table by adding a new column, we use `mutate`. To filter the data table to a subset of rows, we use `filter`. Finally, to subset the data by selecting specific columns, we use `select`.

4.3.1 Adding a column with `mutate`

We want all the necessary information for our analysis to be included in the data table. So the first task is to add the murder rates to our murders data frame. The function `mutate` takes the data frame as a first argument and the name and values of the variable as a second argument using the convention `name = values`. So, to add murder rates, we use:

```
library(dslabs)
data("murders")
murders <- mutate(murders, rate = total / population * 100000)
```

Notice that here we used `total` and `population` inside the function, which are objects that are **not** defined in our workspace. But why don't we get an error?

This is one of **dplyr**'s main features. Functions in this package, such as `mutate`, know to look for variables in the data frame provided in the first argument. In the call to mutate above, `total` will have the values in `murders$total`. This approach makes the code much more readable.

We can see that the new column is added:

```
head(murders)
#>        state abb region population total rate
#> 1    Alabama  AL  South    4779736   135 2.82
#> 2     Alaska  AK   West     710231    19 2.68
#> 3    Arizona  AZ   West    6392017   232 3.63
#> 4   Arkansas  AR  South    2915918    93 3.19
#> 5 California  CA   West   37253956  1257 3.37
#> 6   Colorado  CO   West    5029196    65 1.29
```

Although we have overwritten the original `murders` object, this does not change the object that loaded with `data(murders)`. If we load the `murders` data again, the original will overwrite our mutated version.

4.3.2 Subsetting with `filter`

Now suppose that we want to filter the data table to only show the entries for which the murder rate is lower than 0.71. To do this we use the `filter` function, which takes the data table as the first argument and then the conditional statement as the second. Like `mutate`, we can use the unquoted variable names from `murders` inside the function and it will know we mean the columns and not objects in the workspace.

```
filter(murders, rate <= 0.71)
#>           state abb        region population total  rate
#> 1        Hawaii  HI          West    1360301     7 0.515
#> 2          Iowa  IA North Central    3046355    21 0.689
#> 3 New Hampshire  NH     Northeast    1316470     5 0.380
#> 4  North Dakota  ND North Central     672591     4 0.595
#> 5       Vermont  VT     Northeast     625741     2 0.320
```

4.3.3 Selecting columns with `select`

Although our data table only has six columns, some data tables include hundreds. If we want to view just a few, we can use the **dplyr** `select` function. In the code below we select three columns, assign this to a new object and then filter the new object:

```
new_table <- select(murders, state, region, rate)
filter(new_table, rate <= 0.71)
#>           state        region  rate
#> 1        Hawaii          West 0.515
#> 2          Iowa North Central 0.689
#> 3 New Hampshire     Northeast 0.380
#> 4  North Dakota North Central 0.595
#> 5       Vermont     Northeast 0.320
```

In the call to `select`, the first argument `murders` is an object, but `state`, `region`, and `rate` are variable names.

4.4 Exercises

1. Load the **dplyr** package and the murders dataset.

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

You can add columns using the **dplyr** function `mutate`. This function is aware of the column names and inside the function you can call them unquoted:

```
murders <- mutate(murders, population_in_millions = population / 10^6)
```

We can write `population` rather than `murders$population`. The function `mutate` knows we are grabbing columns from `murders`.

Use the function `mutate` to add a murders column named `rate` with the per 100,000 murder rate as in the example code above. Make sure you redefine `murders` as done in the example code above (murders <- [your code]) so we can keep using this variable.

2. If `rank(x)` gives you the ranks of `x` from lowest to highest, `rank(-x)` gives you the ranks from highest to lowest. Use the function `mutate` to add a column `rank` containing the rank, from highest to lowest murder rate. Make sure you redefine `murders` so we can keep using this variable.

3. With **dplyr**, we can use `select` to show only certain columns. For example, with this code we would only show the states and population sizes:

```
select(murders, state, population) %>% head()
```

Use `select` to show the state names and abbreviations in `murders`. Do not redefine `murders`, just show the results.

4. The **dplyr** function `filter` is used to choose specific rows of the data frame to keep. Unlike `select` which is for columns, `filter` is for rows. For example, you can show just the New York row like this:

```
filter(murders, state == "New York")
```

You can use other logical vectors to filter rows.

Use `filter` to show the top 5 states with the highest murder rates. After we add murder rate and rank, do not change the murders dataset, just show the result. Remember that you can filter based on the `rank` column.

5. We can remove rows using the `!=` operator. For example, to remove Florida, we would do this:

```
no_florida <- filter(murders, state != "Florida")
```

Create a new data frame called `no_south` that removes states from the South region. How many states are in this category? You can use the function `nrow` for this.

6. We can also use `%in%` to filter with **dplyr**. You can therefore see the data from New York and Texas like this:

```
filter(murders, state %in% c("New York", "Texas"))
```

Create a new data frame called `murders_nw` with only the states from the Northeast and the West. How many states are in this category?

7. Suppose you want to live in the Northeast or West **and** want the murder rate to be less than 1. We want to see the data for the states satisfying these options. Note that you can use logical operators with `filter`. Here is an example in which we filter to keep only small states in the Northeast region.

```
filter(murders, population < 5000000 & region == "Northeast")
```

Make sure `murders` has been defined with `rate` and `rank` and still has all states. Create a table called `my_states` that contains rows for states satisfying both the conditions: it is in the Northeast or West and the murder rate is less than 1. Use `select` to show only the state name, the rate, and the rank.

4.5 The pipe: %>%

With **dplyr** we can perform a series of operations, for example `select` and then `filter`, by sending the results of one function to another using what is called the *pipe operator*: `%>%`. Some details are included below.

We wrote code above to show three variables (state, region, rate) for states that have murder rates below 0.71. To do this, we defined the intermediate object `new_table`. In **dplyr** we can write code that looks more like a description of what we want to do without intermediate objects:

$$\text{original data} \rightarrow \text{select} \rightarrow \text{filter}$$

For such an operation, we can use the pipe `%>%`. The code looks like this:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
#>           state        region  rate
#> 1        Hawaii          West 0.515
#> 2          Iowa North Central 0.689
#> 3 New Hampshire     Northeast 0.380
#> 4  North Dakota North Central 0.595
#> 5       Vermont     Northeast 0.320
```

This line of code is equivalent to the two lines of code above. What is going on here?

In general, the pipe *sends* the result of the left side of the pipe to be the first argument of the function on the right side of the pipe. Here is a very simple example:

```
16 %>% sqrt()
#> [1] 4
```

We can continue to pipe values along:

```
16 %>% sqrt() %>% log2()
#> [1] 2
```

The above statement is equivalent to `log2(sqrt(16))`.

Remember that the pipe sends values to the first argument, so we can define other arguments as if the first argument is already defined:

```
16 %>% sqrt() %>% log(base = 2)
#> [1] 2
```

Therefore, when using the pipe with data frames and **dplyr**, we no longer need to specify the required first argument since the **dplyr** functions we have described all take the data as the first argument. In the code we wrote:

```
murders %>% select(state, region, rate) %>% filter(rate <= 0.71)
```

`murders` is the first argument of the `select` function, and the new data frame (formerly `new_table`) is the first argument of the `filter` function.

Note that the pipe works well with functions where the first argument is the input data. Functions in **tidyverse** packages like **dplyr** have this format and can be used easily with the pipe.

4.6 Exercises

1. The pipe `%>%` can be used to perform operations sequentially without having to define intermediate objects. Start by redefining murder to include rate and rank.

```
murders <- mutate(murders, rate =  total / population * 100000,
                  rank = rank(-rate))
```

In the solution to the previous exercise, we did the following:

```
my_states <- filter(murders, region %in% c("Northeast", "West") &
                      rate < 1)

select(my_states, state, rate, rank)
```

The pipe %>% permits us to perform both operations sequentially without having to define an intermediate variable my_states. We therefore could have mutated and selected in the same line like this:

```
mutate(murders, rate =  total / population * 100000,
       rank = rank(-rate)) %>%
  select(state, rate, rank)
```

Notice that select no longer has a data frame as the first argument. The first argument is assumed to be the result of the operation conducted right before the %>%.

Repeat the previous exercise, but now instead of creating a new object, show the result and only include the state, rate, and rank columns. Use a pipe %>% to do this in just one line.

2. Reset murders to the original table by using data(murders). Use a pipe to create a new data frame called my_states that considers only states in the Northeast or West which have a murder rate lower than 1, and contains only the state, rate and rank columns. The pipe should also have four components separated by three %>%. The code should look something like this:

```
my_states <- murders %>%
  mutate SOMETHING %>%
  filter SOMETHING %>%
  select SOMETHING
```

4.7 Summarizing data

An important part of exploratory data analysis is summarizing data. The average and standard deviation are two examples of widely used summary statistics. More informative summaries can often be achieved by first splitting data into groups. In this section, we cover two new **dplyr** verbs that make these computations easier: summarize and group_by. We learn to access resulting values using the pull function.

4.7.1 summarize

The summarize function in **dplyr** provides a way to compute summary statistics with intuitive and readable code. We start with a simple example based on heights. The heights dataset includes heights and sex reported by students in an in-class survey.

```
library(dplyr)
library(dslabs)
data(heights)
```

The following code computes the average and standard deviation for females:

```
s <- heights %>%
  filter(sex == "Female") %>%
  summarize(average = mean(height), standard_deviation = sd(height))
s
#>   average standard_deviation
#> 1    64.9               3.76
```

This takes our original data table as input, filters it to keep only females, and then produces a new summarized table with just the average and the standard deviation of heights. We get to choose the names of the columns of the resulting table. For example, above we decided to use `average` and `standard_deviation`, but we could have used other names just the same.

Because the resulting table stored in `s` is a data frame, we can access the components with the accessor `$`:

```
s$average
#> [1] 64.9
s$standard_deviation
#> [1] 3.76
```

As with most other **dplyr** functions, `summarize` is aware of the variable names and we can use them directly. So when inside the call to the `summarize` function we write `mean(height)`, the function is accessing the column with the name "height" and then computing the average of the resulting numeric vector. We can compute any other summary that operates on vectors and returns a single value. For example, we can add the median, minimum, and maximum heights like this:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(median = median(height), minimum = min(height),
            maximum = max(height))
#>   median minimum maximum
#> 1     65      51      79
```

We can obtain these three values with just one line using the `quantile` function: for example, `quantile(x, c(0,0.5,1))` returns the min (0th percentile), median (50th percentile), and max (100th percentile) of the vector `x`. However, if we attempt to use a function like this that returns two or more values inside `summarize`:

```
heights %>%
  filter(sex == "Female") %>%
  summarize(range = quantile(height, c(0, 0.5, 1)))
```

we will receive an error: `Error: expecting result of length one, got : 2`. With the function `summarize`, we can only call functions that return a single value. In Section 4.12, we will learn how to deal with functions that return more than one value.

For another example of how we can use the `summarize` function, let's compute the average murder rate for the United States. Remember our data table includes total murders and population size for each state and we have already used **dplyr** to add a murder rate column:

```
murders <- murders %>% mutate(rate = total/population*100000)
```

Remember that the US murder rate is **not** the average of the state murder rates:

```
summarize(murders, mean(rate))
#>   mean(rate)
#> 1       2.78
```

This is because in the computation above the small states are given the same weight as the large ones. The US murder rate is the total number of murders in the US divided by the total US population. So the correct computation is:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000)
us_murder_rate
#>   rate
#> 1 3.03
```

This computation counts larger states proportionally to their size which results in a larger value.

4.7.2 `pull`

The `us_murder_rate` object defined above represents just one number. Yet we are storing it in a data frame:

```
class(us_murder_rate)
#> [1] "data.frame"
```

since, as most **dplyr** functions, `summarize` always returns a data frame.

This might be problematic if we want to use this result with functions that require a numeric value. Here we show a useful trick for accessing values stored in data when using pipes: when a data object is piped that object and its columns can be accessed using the `pull` function. To understand what we mean take a look at this line of code:

```
us_murder_rate %>% pull(rate)
#> [1] 3.03
```

This returns the value in the `rate` column of `us_murder_rate` making it equivalent to `us_murder_rate$rate`.

To get a number from the original data table with one line of code we can type:

```
us_murder_rate <- murders %>%
  summarize(rate = sum(total) / sum(population) * 100000) %>%
  pull(rate)

us_murder_rate
#> [1] 3.03
```

which is now a numeric:

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

4.7.3 Group then summarize with `group_by`

A common operation in data exploration is to first split data into groups and then compute summaries for each group. For example, we may want to compute the average and standard deviation for men's and women's heights separately. The `group_by` function helps us do this.

If we type this:

```
heights %>% group_by(sex)
#> # A tibble: 1,050 x 2
#> # Groups:   sex [2]
#>   sex   height
#>   <fct>  <dbl>
#> 1 Male      75
#> 2 Male      70
#> 3 Male      68
#> 4 Male      74
#> 5 Male      61
#> # ... with 1,045 more rows
```

The result does not look very different from `heights`, except we see `Groups: sex [2]` when we print the object. Although not immediately obvious from its appearance, this is now a special data frame called a *grouped data frame* and **dplyr** functions, in particular `summarize`, will behave differently when acting on this object. Conceptually, you can think of this table as many tables, with the same columns but not necessarily the same number of rows, stacked together in one object. When we summarize the data after grouping, this is what happens:

```
heights %>%
  group_by(sex) %>%
  summarize(average = mean(height), standard_deviation = sd(height))
#> # A tibble: 2 x 3
#>   sex    average standard_deviation
#>   <fct>    <dbl>              <dbl>
#> 1 Female    64.9               3.76
#> 2 Male      69.3               3.61
```

The `summarize` function applies the summarization to each group separately.

For another example, let's compute the median murder rate in the four regions of the country:

```
murders %>%
  group_by(region) %>%
  summarize(median_rate = median(rate))
```

```
#> # A tibble: 4 x 2
#>   region        median_rate
#>   <fct>               <dbl>
#> 1 Northeast            1.80
#> 2 South                3.40
#> 3 North Central        1.97
#> 4 West                 1.29
```

4.8 Sorting data frames

When examining a dataset, it is often convenient to sort the table by the different columns. We know about the `order` and `sort` function, but for ordering entire tables, the **dplyr** function `arrange` is useful. For example, here we order the states by population size:

```
murders %>%
  arrange(population) %>%
  head()
#>                  state abb        region population total   rate
#> 1              Wyoming  WY          West     563626     5  0.887
#> 2 District of Columbia  DC         South     601723    99 16.453
#> 3              Vermont  VT     Northeast     625741     2  0.320
#> 4         North Dakota  ND North Central     672591     4  0.595
#> 5               Alaska  AK          West     710231    19  2.675
#> 6         South Dakota  SD North Central     814180     8  0.983
```

With `arrange` we get to decide which column to sort by. To see the states by population, from smallest to largest, we arrange by `rate` instead:

```
murders %>%
  arrange(rate) %>%
  head()
#>           state abb        region population total  rate
#> 1       Vermont  VT     Northeast     625741     2 0.320
#> 2 New Hampshire  NH     Northeast    1316470     5 0.380
#> 3        Hawaii  HI          West    1360301     7 0.515
#> 4  North Dakota  ND North Central     672591     4 0.595
#> 5          Iowa  IA North Central    3046355    21 0.689
#> 6         Idaho  ID          West    1567582    12 0.766
```

Note that the default behavior is to order in ascending order. In **dplyr**, the function `desc` transforms a vector so that it is in descending order. To sort the table in descending order, we can type:

```
murders %>%
  arrange(desc(rate))
```

4.8.1 Nested sorting

If we are ordering by a column with ties, we can use a second column to break the tie. Similarly, a third column can be used to break ties between first and second and so on. Here we order by `region`, then within region we order by murder rate:

```
murders %>%
  arrange(region, rate) %>%
  head()
#>            state abb    region population total  rate
#> 1        Vermont  VT Northeast     625741     2 0.320
#> 2 New Hampshire  NH Northeast    1316470     5 0.380
#> 3          Maine  ME Northeast    1328361    11 0.828
#> 4   Rhode Island  RI Northeast    1052567    16 1.520
#> 5 Massachusetts  MA Northeast    6547629   118 1.802
#> 6       New York  NY Northeast   19378102   517 2.668
```

4.8.2 The top n

In the code above, we have used the function `head` to avoid having the page fill up with the entire dataset. If we want to see a larger proportion, we can use the `top_n` function. This function takes a data frame as it's first argument, the number of rows to show in the second, and the variable to filter by in the third. Here is an example of how to see the top 5 rows:

```
murders %>% top_n(5, rate)
#>                  state abb        region population total  rate
#> 1 District of Columbia  DC         South     601723    99 16.45
#> 2            Louisiana  LA         South    4533372   351  7.74
#> 3             Maryland  MD         South    5773552   293  5.07
#> 4             Missouri  MO North Central    5988927   321  5.36
#> 5       South Carolina  SC         South    4625364   207  4.48
```

Note that rows are not sorted by `rate`, only filtered. If we want to sort, we need to use `arrange`. Note that if the third argument is left blank, `top_n`, filters by the last column.

4.9 Exercises

For these exercises, we will be using the data from the survey collected by the United States National Center for Health Statistics (NCHS). This center has conducted a series of health and nutrition surveys since the 1960's. Starting in 1999, about 5,000 individuals of all ages have been interviewed every year and they complete the health examination component of the survey. Part of the data is made available via the **NHANES** package. Once you install the **NHANES** package, you can load the data like this:

```
library(NHANES)
data(NHANES)
```

The **NHANES** data has many missing values. Remember that the main summarization function in R will return `NA` if any of the entries of the input vector is an `NA`. Here is an example:

```
library(dslabs)
data(na_example)
mean(na_example)
#> [1] NA
sd(na_example)
#> [1] NA
```

To ignore the `NA`s we can use the `na.rm` argument:

```
mean(na_example, na.rm = TRUE)
#> [1] 2.3
sd(na_example, na.rm = TRUE)
#> [1] 1.22
```

Let's now explore the NHANES data.

1. We will provide some basic facts about blood pressure. First let's select a group to set the standard. We will use 20-to-29-year-old females. `AgeDecade` is a categorical variable with these ages. Note that the category is coded like " 20-29", with a space in front! What is the average and standard deviation of systolic blood pressure as saved in the `BPSysAve` variable? Save it to a variable called `ref`.

Hint: Use `filter` and `summarize` and use the `na.rm = TRUE` argument when computing the average and standard deviation. You can also filter the NA values using `filter`.

2. Using a pipe, assign the average to a numeric variable `ref_avg`. Hint: Use the code similar to above and then `pull`.

3. Now report the min and max values for the same group.

4. Compute the average and standard deviation for females, but for each age group separately rather than a selected decade as in question 1. Note that the age groups are defined by `AgeDecade`. Hint: rather than filtering by age and gender, filter by `Gender` and then use `group_by`.

5. Repeat exercise 4 for males.

6. We can actually combine both summaries for exercises 4 and 5 into one line of code. This is because `group_by` permits us to group by more than one variable. Obtain one big summary table using `group_by(AgeDecade, Gender)`.

7. For males between the ages of 40-49, compare systolic blood pressure across race as reported in the `Race1` variable. Order the resulting table from lowest to highest average systolic blood pressure.

4.10 Tibbles

Tidy data must be stored in data frames. We introduced the data frame in Section 2.4.1 and have been using the `murders` data frame throughout the book. In Section 4.7.3 we introduced the `group_by` function, which permits stratifying data before computing summary statistics. But where is the group information stored in the data frame?

```
murders %>% group_by(region)
#> # A tibble: 51 x 6
#> # Groups:   region [4]
#>   state      abb   region population total  rate
#>   <chr>      <chr> <fct>       <dbl> <dbl> <dbl>
#> 1 Alabama    AL    South     4779736   135  2.82
#> 2 Alaska     AK    West       710231    19  2.68
#> 3 Arizona    AZ    West      6392017   232  3.63
#> 4 Arkansas   AR    South     2915918    93  3.19
#> 5 California CA    West     37253956  1257  3.37
#> # ... with 46 more rows
```

Notice that there are no columns with this information. But, if you look closely at the output above, you see the line `A tibble` followd by dimensions. We can learn the class of the returned object using:

```
murders %>% group_by(region) %>% class()
#> [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

The `tbl`, pronounced tibble, is a special kind of data frame. The functions `group_by` and `summarize` always return this type of data frame. The `group_by` function returns a special kind of `tbl`, the `grouped_df`. We will say more about these later. For consistency, the **dplyr** manipulation verbs (`select`, `filter`, `mutate`, and `arrange`) preserve the class of the input: if they receive a regular data frame they return a regular data frame, while if they receive a tibble they return a tibble. But tibbles are the preferred format in the tidyverse and as a result tidyverse functions that produce a data frame from scratch return a tibble. For example, in Chapter 5 we will see that tidyverse functions used to import data create tibbles.

Tibbles are very similar to data frames. In fact, you can think of them as a modern version of data frames. Nonetheless there are three important differences which we describe in the next.

4.10.1 Tibbles display better

The print method for tibbles is more readable than that of a data frame. To see this, compare the outputs of typing `murders` and the output of murders if we convert it to a tibble. We can do this using `as_tibble(murders)`. If using RStudio, output for a tibble adjusts to your window size. To see this, change the width of your R console and notice how more/less columns are shown.

4.10.2 Subsets of tibbles are tibbles

If you subset the columns of a data frame, you may get back an object that is not a data frame, such as a vector or scalar. For example:

```
class(murders[,4])
#> [1] "numeric"
```

is not a data frame. With tibbles this does not happen:

```
class(as_tibble(murders)[,4])
#> [1] "tbl_df"     "tbl"        "data.frame"
```

This is useful in the tidyverse since functions require data frames as input.

With tibbles, if you want to access the vector that defines a column, and not get back a data frame, you need to use the accessor `$`:

```
class(as_tibble(murders)$population)
#> [1] "numeric"
```

A related feature is that tibbles will give you a warning if you try to access a column that does not exist. If we accidentally write `Population` instead of `population` this:

```
murders$Population
#> NULL
```

returns a `NULL` with no warning, which can make it harder to debug. In contrast, if we try this with a tibble we get an informative warning:

```
as_tibble(murders)$Population
#> Warning: Unknown or uninitialised column: 'Population'.
#> NULL
```

4.10.3 Tibbles can have complex entries

While data frame columns need to be vectors of numbers, strings, or logical values, tibbles can have more complex objects, such as lists or functions. Also, we can create tibbles with functions:

```
tibble(id = c(1, 2, 3), func = c(mean, median, sd))
#> # A tibble: 3 x 2
#>      id func
#>   <dbl> <list>
#> 1     1 <fn>
#> 2     2 <fn>
#> 3     3 <fn>
```

4.10.4 Tibbles can be grouped

The function `group_by` returns a special kind of tibble: a grouped tibble. This class stores information that lets you know which rows are in which groups. The tidyverse functions, in particular the `summarize` function, are aware of the group information.

4.10.5 Create a tibble using `tibble` instead of `data.frame`

It is sometimes useful for us to create our own data frames. To create a data frame in the tibble format, you can do this by using the `tibble` function.

```
grades <- tibble(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
```

Note that base R (without packages loaded) has a function with a very similar name, `data.frame`, that can be used to create a regular data frame rather than a tibble. One other important difference is that by default `data.frame` coerces characters into factors without providing a warning or message:

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90))
class(grades$names)
#> [1] "factor"
```

To avoid this, we use the rather cumbersome argument `stringsAsFactors`:

```
grades <- data.frame(names = c("John", "Juan", "Jean", "Yao"),
                     exam_1 = c(95, 80, 90, 85),
                     exam_2 = c(90, 85, 85, 90),
                     stringsAsFactors = FALSE)
class(grades$names)
#> [1] "character"
```

To convert a regular data frame to a tibble, you can use the `as_tibble` function.

```
as_tibble(grades) %>% class()
#> [1] "tbl_df"     "tbl"        "data.frame"
```

4.11 The dot operator

One of the advantages of using the pipe `%>%` is that we do not have to keep naming new objects as we manipulate the data frame. As a quick reminder, if we want to compute the median murder rate for states in the southern states, instead of typing:

```
tab_1 <- filter(murders, region == "South")
tab_2 <- mutate(tab_1, rate = total / population * 10^5)
rates <- tab_2$rate
median(rates)
#> [1] 3.4
```

We can avoid defining any new intermediate objects by instead typing:

```
filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  summarize(median = median(rate)) %>%
  pull(median)
#> [1] 3.4
```

We can do this because each of these functions takes a data frame as the first argument. But what if we want to access a component of the data frame. For example, what if the `pull` function was not available and we wanted to access `tab_2$rate`? What data frame name would we use? The answer is the dot operator.

For example to access the rate vector without the `pull` function we could use

```
rates <-filter(murders, region == "South") %>%
  mutate(rate = total / population * 10^5) %>%
  .$rate
median(rates)
#> [1] 3.4
```

In the next section, we will see other instances in which using the `.` is useful.

4.12 `do`

The tidyverse functions know how to interpret grouped tibbles. Furthermore, to facilitate stringing commands through the pipe `%>%`, tidyverse functions consistently return data frames, since this assures that the output of a function is accepted as the input of another. But most R functions do not recognize grouped tibbles nor do they return data frames. The `quantile` function is an example we described in Section 4.7.1. The `do` function serves as a bridge between R functions such as `quantile` and the tidyverse. The `do` function understands grouped tibbles and always returns a data frame.

In Section 4.7.1, we noted that if we attempt to use `quantile` to obtain the min, median and max in one call, we will receive an error: `Error: expecting result of length one, got : 2`.

```
data(heights)
heights %>%
  filter(sex == "Female") %>%
  summarize(range = quantile(height, c(0, 0.5, 1)))
```

We can use the `do` function to fix this.

First we have to write a function that fits into the tidyverse approach: that is, it receives a data frame and returns a data frame.

```
my_summary <- function(dat){
  x <- quantile(dat$height, c(0, 0.5, 1))
  tibble(min = x[1], median = x[2], max = x[3])
}
```

We can now apply the function to the heights dataset to obtain the summaries:

```
heights %>%
  group_by(sex) %>%
  my_summary
#> # A tibble: 1 x 3
#>     min median   max
#>   <dbl>  <dbl> <dbl>
#> 1    50   68.5  82.7
```

But this is not what we want. We want a summary for each sex and the code returned just one summary. This is because `my_summary` is not part of the tidyverse and does not know how to handled grouped tibbles. `do` makes this connection:

```
heights %>%
  group_by(sex) %>%
  do(my_summary(.))
#> # A tibble: 2 x 4
#> # Groups:   sex [2]
#>   sex      min median   max
#>   <fct>  <dbl>  <dbl> <dbl>
#> 1 Female    51   65.0  79
#> 2 Male      50   69    82.7
```

Note that here we need to use the dot operator. The tibble created by `group_by` is piped to `do`. Within the call to `do`, the name of this tibble is `.` and we want to send it to `my_summary`. If you do not use the dot, then `my_summary` has ___no argument and returns an error telling us that `argument "dat"` is missing. You can see the error by typing:

```
heights %>%
  group_by(sex) %>%
  do(my_summary())
```

If you do not use the parenthesis, then the function is not executed and instead `do` tries to return the function. This gives an error because `do` must always return a data frame. You can see the error by typing:

```
heights %>%
  group_by(sex) %>%
  do(my_summary)
```

4.13 The purrr package

In Section 3.5 we learned about the `sapply` function, which permitted us to apply the same function to each element of a vector. We constructed a function and used `sapply` to compute the sum of the first `n` integers for several values of `n` like this:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

This type of operation, applying the same function or procedure to elements of an object, is quite common in data analysis. The **purrr** package includes functions similar to `sapply` but that better interact with other tidyverse functions. The main advantage is that we can better control the output type of functions. In contrast, `sapply` can return several different object types; for example, we might expect a numeric result from a line of code, but `sapply` might convert our result to character under some circumstances. **purrr** functions will never do this: they will return objects of a specified type or return an error if this is not possible.

The first **purrr** function we will learn is `map`, which works very similar to `sapply` but always, without exception, returns a list:

```
library(purrr)
s_n <- map(n, compute_s_n)
class(s_n)
#> [1] "list"
```

If we want a numeric vector, we can instead use `map_dbl` which always returns a vector of numeric values.

```
s_n <- map_dbl(n, compute_s_n)
class(s_n)
#> [1] "numeric"
```

This produces the same results as the `sapply` call shown above.

A particularly useful **purrr** function for interacting with the rest of the tidyverse is `map_df`, which always returns a tibble data frame. However, the function being called needs to return a vector or a list with names. For this reason, the following code would result in a `Argument 1 must have names` error:

```
s_n <- map_df(n, compute_s_n)
```

We need to change the function to make this work:

```
compute_s_n <- function(n){
  x <- 1:n
  tibble(sum = sum(x))
}
s_n <- map_df(n, compute_s_n)
```

The **purrr** package provides much more functionality not covered here. For more details you can consult this online resource.

4.14 Tidyverse conditionals

A typical data analysis will often involve one or more conditional operations. In Section 3.1 we described the `ifelse` function, which we will use extensively in this book. In this section we present two **dplyr** functions that provide further functionality for performing conditional operations.

4.14.1 `case_when`

The `case_when` function is useful for vectorizing conditional statements. It is similar to `ifelse` but can output any number of values, as opposed to just `TRUE` or `FALSE`. Here is an example splitting numbers into negative, positive, and 0:

```
x <- c(-2, -1, 0, 1, 2)
case_when(x < 0 ~ "Negative", x > 0 ~ "Positive", TRUE ~ "Zero")
#> [1] "Negative" "Negative" "Zero"     "Positive" "Positive"
```

A common use for this function is to define categorical variables based on existing variables. For example, suppose we want to compare the murder rates in three groups of states: *New England*, *West Coast*, *South*, and *other*. For each state, we need to ask if it is in New England, if it is not we ask if it is in the West Coast, if not we ask if it is in the South, and if not we assign *other*. Here is how we use `case_when` to do this:

```
murders %>%
  mutate(group = case_when(
    abb %in% c("ME", "NH", "VT", "MA", "RI", "CT") ~ "New England",
    abb %in% c("WA", "OR", "CA") ~ "West Coast",
    region == "South" ~ "South",
    TRUE ~ "Other")) %>%
  group_by(group) %>%
  summarize(rate = sum(total) / sum(population) * 10^5)
#> # A tibble: 4 x 2
#>   group        rate
#>   <chr>       <dbl>
#> 1 New England  1.72
#> 2 Other        2.71
#> 3 South        3.63
#> 4 West Coast   2.90
```

4.14.2 between

A common operation in data analysis is to determine if a value falls inside an interval. We can check this using conditionals. For example to check if the elements of a vector `x` are between `a` and `b` we can type

```
x >= a & x <= b
```

However, this can become cumbersome, especially within the tidyverse approach. The `between` function performs the same operation.

```
between(x, a, b)
```

4.15 Exercises

1. Load the `murders` dataset. Which of the following is true?

   a. `murders` is in tidy format and is stored in a tibble.
   b. `murders` is in tidy format and is stored in a data frame.
   c. `murders` is not in tidy format and is stored in a tibble.
   d. `murders` is not in tidy format and is stored in a data frame.

2. Use `as_tibble` to convert the `murders` data table into a tibble and save it in an object called `murders_tibble`.

3. Use the `group_by` function to convert murders into a tibble that is grouped by region.

4. Write tidyverse code that is equivalent to this code:

```
exp(mean(log(murders$population)))
```

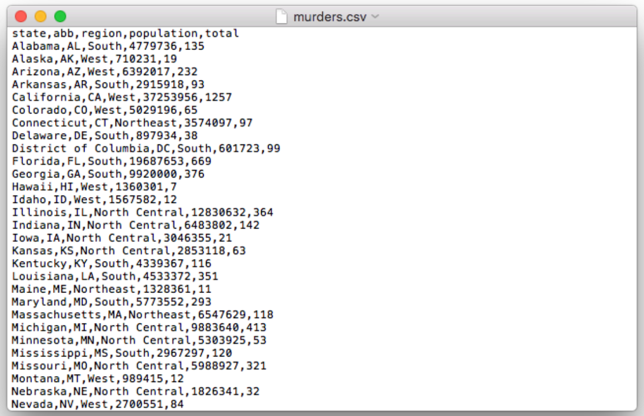
Write it using the pipe so that each function is called without arguments. Use the dot operator to access the population. Hint: The code should start with `murders %>%`.

5. Use the `map_df` to create a data frame with three columns named `n`, `s_n`, and `s_n_2`. The first column should contain the numbers 1 through 100. The second and third columns should each contain the sum of 1 through n with n the row number.

5 Importing data

We have been using data sets already stored as R objects. A data scientist will rarely have such luck and will have to import data into R from either a file, a database, or other sources. Currently, one of the most common ways of storing and sharing data for analysis is through electronic spreadsheets. A spreadsheet stores data in rows and columns. It is basically a file version of a data frame. When saving such a table to a computer file, one needs a way to define when a new row or column ends and the other begins. This in turn defines the cells in which single values are stored.

When creating spreadsheets with text files, like the ones created with a simple text editor, a new row is defined with return and columns are separated with some predefined special character. The most common characters are comma (`,`), semicolon (`;`), space ( ), and tab (a preset number of spaces or `\t`). Here is an example of what a comma separated file looks like if we open it with a basic text editor:



```
state,abb,region,population,total
Alabama,AL,South,4779736,135
Alaska,AK,West,710231,19
Arizona,AZ,West,6392017,232
Arkansas,AR,South,2915918,93
California,CA,West,37253956,1257
Colorado,CO,West,5029196,65
Connecticut,CT,Northeast,3574097,97
Delaware,DE,South,897934,38
District of Columbia,DC,South,601723,99
Florida,FL,South,19687653,669
Georgia,GA,South,9920000,376
Hawaii,HI,West,1360301,7
Idaho,ID,West,1567582,12
Illinois,IL,North Central,12830632,364
Indiana,IN,North Central,6483802,142
Iowa,IA,North Central,3046355,21
Kansas,KS,North Central,2853118,63
Kentucky,KY,South,4339367,116
Louisiana,LA,South,4533372,351
Maine,ME,Northeast,1328361,11
Maryland,MD,South,5773552,293
Massachusetts,MA,Northeast,6547629,118
Michigan,MI,North Central,9883640,413
Minnesota,MN,North Central,5303925,53
Mississippi,MS,South,2967297,120
Missouri,MO,North Central,5988927,321
Montana,MT,West,989415,12
Nebraska,NE,North Central,1826341,32
Nevada,NV,West,2700551,84
```

The first row contains column names rather than data. We call this a *header*, and when we read-in data from a spreadsheet it is important to know if the file has a header or not. Most reading functions assume there is a header. To know if the file has a header, it helps to look at the file before trying to read it. This can be done with a text editor or with RStudio. In RStudio, we can do this by either opening the file in the editor or navigating to the file location, double clicking on the file, and hitting *View File*.

However, not all spreadsheet files are in a text format. Google Sheets, which are rendered on a browser, are an example. Another example is the proprietary format used by Microsoft Excel. These can't be viewed with a text editor. Despite this, due to the widespread use of Microsoft Excel software, this format is widely used.

We start this chapter by describing the difference between text (ASCII), Unicode, and binary files and how this affects how we import them. We then explain the concepts of file paths and working directories, which are essential to understand how to import data effectively. We then introduce the **readr** and **readxl** package and the functions that are available to import spreadsheets into R. Finally, we provide some recommendations on how to store and organize data in files. More complex challenges such as extracting data from web pages or PDF documents are left for the Data Wrangling part of the book.

5.1 Paths and the working directory

The first step when importing data from a spreadsheet is to locate the file containing the data. Although we do not recommend it, you can use an approach similar to what you do to open files in Microsoft Excel by clicking on the RStudio "File" menu, clicking "Import Dataset", then clicking through folders until you find the file. We want to be able to write code rather than use the point-and-click approach. The keys and concepts we need to learn to do this are described in detail in the Productivity Tools part of this book. Here we provide an overview of the very basics.

The main challenge in this first step is that we need to let the R functions doing the importing know where to look for the file containing the data. The simplest way to do this is to have a copy of the file in the folder in which the importing functions look by default. Once we do this, all we have to supply to the importing function is the filename.

A spreadsheet containing the US murders data is included as part of the **dslabs** package. Finding this file is not straightforward, but the following lines of code copy the file to the folder in which R looks in by default. We explain how these lines work below.

```r
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

This code does not read the data into R, it just copies a file. But once the file is copied, we can import the data with a simple line of code. Here we use the `read_csv` function from the **readr** package, which is part of the tidyverse.

```r
library(tidyverse)
dat <- read_csv(filename)
```

The data is imported and stored in `dat`. The rest of this section defines some important concepts and provides an overview of how we write code that tells R how to find the files we want to import. Chapter 36 provides more details on this topic.

5.1.1 The filesystem

You can think of your computer's filesystem as a series of nested folders, each containing other folders and files. Data scientists refer to folders as *directories*. We refer to the folder that contains all other folders as the *root directory*. We refer to the directory in which we are currently located as the *working directory*. The working directory therefore changes as you move through folders: think of it as your current location.

5.1.2 Relative and full paths

The *path* of a file is a list of directory names that can be thought of as instructions on what folders to click on, and in what order, to find the file. If these instructions are for finding the file from the root directory we refer to it as the *full path*. If the instructions are for finding the file starting in the working directory we refer to it as a *relative path*. Section 36.3 provides more details on this topic.

To see an example of a full path on your system type the following:

```
system.file(package = "dslabs")
```

The strings separated by slashes are the directory names. The first slash represents the root directory and we know this is a full path because it starts with a slash. If the first directory name appears without a slash in front, then the path is assumed to be relative. We can use the function `list.files` to see examples of relative paths.

```
dir <- system.file(package = "dslabs")
list.files(path = dir)
#>  [1] "data"        "DESCRIPTION" "extdata"     "help"
#>  [5] "html"        "INDEX"       "Meta"        "NAMESPACE"
#>  [9] "R"           "script"
```

These relative paths give us the location of the files or directories if we start in the directory with the full path. For example, the full path to the `help` directory in the example above is `/Library/Frameworks/R.framework/Versions/3.5/Resources/library/dslabs/help`.

Note: You will probably not make much use of the `system.file` function in your day-to-day data analysis work. We introduce it in this section because it facilitates the sharing of spreadsheets by including them in the **dslabs** package. You will rarely have the luxury of data being included in packages you already have installed. However, you will frequently need to navigate full and relative paths and import spreadsheet formatted data.

5.1.3 The working directory

We highly recommend only writing relative paths in your code. The reason is that full paths are unique to your computer and you want your code to be portable. You can get the full path of your working directory without writing out explicitly by using the `getwd` function.

```
wd <- getwd()
```

If you need to change your working directory, you can use the function `setwd` or you can change it through RStudio by clicking on "Session".

5.1.4 Generating path names

Another example of obtaining a full path without writing out explicitly was given above when we created the object `fullpath` like this:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
```

The function `system.file` provides the full path of the folder containing all the files and directories relevant to the package specified by the `package` argument. By exploring the directories in `dir` we find that the `extdata` contains the file we want:

```
dir <- system.file(package = "dslabs")
filename %in% list.files(file.path(dir, "extdata"))
#> [1] TRUE
```

The `system.file` function permits us to provide a subdirectory as a first argument, so we can obtain the fullpath of the `extdata` directory like this:

```
dir <- system.file("extdata", package = "dslabs")
```

The function `file.path` is used to combine directory names to produce the full path of the file we want to import.

```
fullpath <- file.path(dir, filename)
```

5.1.5 Copying files using paths

The final line of code we used to copy the file into our home directory used the function `file.copy`. This function takes two arguments: the file to copy and the name to give it in the new directory.

```
file.copy(fullpath, "murders.csv")
#> [1] TRUE
```

If a file is copied successfully, the `file.copy` function returns `TRUE`. Note that we are giving the file the same name, `murders.csv`, but we could have named it anything. Also note that by not starting the string with a slash, R assumes this is a relative path and copies the file to the working directory.

You should be able to see the file in your working directory and can check by using:

```
list.files()
```

5.2 The readr and readxl packages

In this section we introduce the main tidyverse data importing functions. We will use the `murders.csv` file provided by the **dslabs** package as an example. To simplify the illustration we will copy the file to our working directory using the following code:

```
filename <- "murders.csv"
dir <- system.file("extdata", package = "dslabs")
fullpath <- file.path(dir, filename)
file.copy(fullpath, "murders.csv")
```

5.2.1 readr

The **readr** library includes functions for reading data stored in text file spreadsheets into R. **readr** is part of the **tidyverse** package, or you can load it directly:

```
library(readr)
```

The following functions are available to read-in spreadsheets:

Function	Format	Typical suffix
read_table	white space separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must define delimiter	txt

Although the suffix usually tells us what type of file it is, there is no guarantee that these always match. We can open the file to take a look or use the function `read_lines` to look at a few lines:

```
read_lines("murders.csv", n_max = 3)
#> [1] "state,abb,region,population,total"
#> [2] "Alabama,AL,South,4779736,135"
#> [3] "Alaska,AK,West,710231,19"
```

This also shows that there is a header. Now we are ready to read-in the data into R. From the .csv suffix and the peek at the file, we know to use `read_csv`:

```
dat <- read_csv(filename)
#> Parsed with column specification:
#> cols(
#>   state = col_character(),
#>   abb = col_character(),
#>   region = col_character(),
#>   population = col_double(),
#>   total = col_double()
#> )
```

Note that we receive a message letting us know what data types were used for each column. Also note that `dat` is a `tibble`, not just a data frame. This is because `read_csv` is a **tidyverse** parser. We can confirm that the data has in fact been read-in with:

```
View(dat)
```

Finally, note that we can also use the full path for the file:

```
dat <- read_csv(fullpath)
```

5.2.2 readxl

You can load the **readxl** package using

```
library(readxl)
```

The package provides functions to read-in Microsoft Excel formats:

Function	Format	Typical suffix
read_excel	auto detect the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new format	xlsx

The Microsoft Excel formats permit you to have more than one spreadsheet in one file. These are referred to as *sheets*. The functions listed above read the first sheet by default, but we can also read the others. The `excel_sheets` function gives us the names of all the sheets in an Excel file. These names can then be passed to the `sheet` argument in the three functions above to read sheets other than the first.

5.3 Exercises

1. Use the `read_csv` function to read each of the files that the following code saves in the `files` object:

```
path <- system.file("extdata", package = "dslabs")
files <- list.files(path)
files
```

2. Note that the last one, the `olive` file, gives us a warning. This is because the first line of the file is missing the header for the first column.

Read the help file for `read_csv` to figure out how to read in the file without reading this header. If you skip the header, you should not get this warning. Save the result to an object called `dat`.

3. A problem with the previous approach is that we don't know what the columns represent. Type:

```
names(dat)
```

to see that the names are not informative.

Use the `readLines` function to read in just the first line (we later learn how to extract values from the output).

5.4 Downloading files

Another common place for data to reside is on the internet. When these data are in files, we can download them and then import them or even read them directly from the web. For example, we note that because our **dslabs** package is on GitHub, the file we downloaded with the package has a url:

```
url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/
extdata/murders.csv"
```

The `read_csv` file can read these files directly:

```
dat <- read_csv(url)
```

If you want to have a local copy of the file, you can use the `download.file` function:

```
download.file(url, "murders.csv")
```

This will download the file and save it on your system with the name `murders.csv`. You can use any name here, not necessarily `murders.csv`. Note that when using `download.file` you should be careful as it will overwrite existing files without warning.

Two functions that are sometimes useful when downloading data from the internet are `tempdir` and `tempfile`. The first creates a directory with a random name that is very likely to be unique. Similarly, `tempfile` creates a character string, not a file, that is likely to be a unique filename. So you can run a command like this which erases the temporary file once it imports the data:

```
tmp_filename <- tempfile()
download.file(url, tmp_filename)
dat <- read_csv(tmp_filename)
file.remove(tmp_filename)
```

5.5 R-base importing functions

R-base also provides import functions. These have similar names to those in the **tidyverse**, for example `read.table`, `read.csv` and `read.delim`. However, there are a couple of important differences. To show this we read-in the data with an R-base function:

```
dat2 <- read.csv(filename)
```

An important difference is that the characters are converted to factors:

```
class(dat2$abb)
#> [1] "factor"
class(dat2$region)
#> [1] "factor"
```

This can be avoided by setting the argument `stringsAsFactors` to `FALSE`.

```
dat <- read.csv("murders.csv", stringsAsFactors = FALSE)
class(dat$state)
#> [1] "character"
```

In our experience this can be a cause for confusion since a variable that was saved as characters in file is converted to factors regardless of what the variable represents. In fact, we **highly** recommend setting `stringsAsFactors=FALSE` to be your default approach when using the R-base parsers. You can easily convert the desired columns to factors after importing data.

5.5.1 `scan`

When reading in spreadsheets many things can go wrong. The file might have a multiline header, be missing cells, or it might use an unexpected encoding[1]. We recommend you read this post about common issues found here: https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/.

With experience you will learn how to deal with different challenges. Carefully reading the help files for the functions discussed here will be useful. Two other functions that are helpful are `scan`. With scan you can read-in each cell of a file. Here is an example:

```
path <- system.file("extdata", package = "dslabs")
filename <- "murders.csv"
x <- scan(file.path(path, filename), sep=",", what = "c")
x[1:10]
#>  [1] "state"      "abb"        "region"     "population" "total"
#>  [6] "Alabama"    "AL"         "South"      "4779736"    "135"
```

Note that the tidyverse provides `read_lines`, a similarly useful function.

[1] https://en.wikipedia.org/wiki/Character_encoding

5.6 Text versus binary files

For data science purposes, files can generally be classified into two categories: text files (also known as ASCII files) and binary files. You have already worked with text files. All your R scripts are text files and so are the R markdown files used to create this book. The csv tables you have read are also text files. One big advantage of these files is that we can easily "look" at them without having to purchase any kind of special software or follow complicated instructions. Any text editor can be used to examine a text file, including freely available editors such as RStudio, Notepad, textEdit, vi, emacs, nano, and pico. To see this, try opening a csv file using the "Open file" RStudio tool. You should be able to see the content right on your editor. However, if you try to open, say, an Excel xls file, jpg or png file, you will not be able to see anything immediately useful. These are binary files. Excel files are actually compressed folders with several text files inside. But the main distinction here is that text files can be easily examined.

Although R includes tools for reading widely used binary files, such as xls files, in general you will want to find data sets stored in text files. Similarly, when sharing data you want to make it available as text files as long as storage is not an issue (binary files are much more efficient at saving space on your disk). In general, plain-text formats make it easier to share data since commercial software is not required for working with the data.

Extracting data from a spreadsheet stored as a text file is perhaps the easiest way to bring data from a file to an R session. Unfortunately, spreadsheets are not always available and the fact that you can look at text files does not necessarily imply that extracting data from them will be straightforward. In the Data Wrangling part of the book we learn to extract data from more complex text files such as html files.

5.7 Unicode versus ASCII

A pitfall in data science is assuming a file is an ASCII text file when, in fact, it is something else that can look a lot like an ASCII text file: a Unicode text file.

To understand the difference between these, remember that everything on a computer needs to eventually be converted to 0s and 1s. ASCII is an *encoding* that maps characters to numbers. ASCII uses 7 bits (0s and 1s) which results in $2^7 = 128$ unique items, enough to encode all the characters on an English language keyboard. However, other languages use characters not included in this encoding. For example, the é in México is not encoded by ASCII. For this reason, a new encoding, using more than 7 bits, was defined: Unicode. When using Unicode, one can chose between 8, 16, and 32 bits abbreviated UTF-8, UTF-16, and UTF-32 respectively. RStudio actually defaults to UTF-8 encoding.

Although we do not go into the details of how to deal with the different encodings here, it is important that you know these different encodings exist so that you can better diagnose a problem if you encounter it. One way problems manifest themselves is when you see "weird looking" characters you were not expecting. This StackOverflow discussion is an example: https://stackoverflow.com/questions/18789330/r-on-windows-character-encoding-hell.

5.8 Organizing data with spreadsheets

Although there are R packages designed to read this format, if you are choosing a file format to save your own data, you generally want to avoid Microsoft Excel. We recommend Google Sheets as a free software tool for organizing data. We provide more recommendations in the section Data Organization with Spreadsheets. This book focuses on data analysis. Yet often a data scientist needs to collect data or work with others collecting data. Filling out a spreadsheet by hand is a practice we highly discourage and instead recommend that the process be automatized as much as possible. But sometimes you just have to do it. In this section, we provide recommendations on how to store data in a spreadsheet. We summarize a paper by Karl Broman and Kara Woo[2]. Below are their general recommendations. Please read the paper for important details.

- **Be Consistent -** Before you commence entering data, have a plan. Once you have a plan, be consistent and stick to it.
- **Choose Good Names for Things -** You want the names you pick for objects, files, and directories to be memorable, easy to spell, and descriptive. This is actually a hard balance to achieve and it does require time and thought. One important rule to follow is **do not use spaces**, use underscores `_` or dashes instead `-`. Also, avoid symbols; stick to letters and numbers.
- **Write Dates as YYYY-MM-DD -** To avoid confusion, we strongly recommend using this global ISO 8601 standard.
- **No Empty Cells -** Fill in all cells and use some common code for missing data.
- **Put Just One Thing in a Cell -** It is better to add columns to store the extra information rather than having more than one piece of information in one cell.
- **Make It a Rectangle -** The spreadsheet should be a rectangle.
- **Create a Data Dictionary -** If you need to explain things, such as what the columns are or what the labels used for categorical variables are, do this in a separate file.
- **No Calculations in the Raw Data Files -** Excel permits you to perform calculations. Do not make this part of your spreadsheet. Code for calculations should be in a script.
- **Do Not Use Font Color or Highlighting as Data -** Most import functions are not able to import this information. Encode this information as a variable instead.
- **Make Backups -** Make regular backups of your data.
- **Use Data Validation to Avoid Errors -** Leverage the tools in your spreadsheet software so that the process is as error-free and repetitive-stress-injury-free as possible.
- **Save the Data as Text Files -** Save files for sharing in comma or tab delimited format.

5.9 Exercises

1. Pick a measurement you can take on a regular basis. For example, your daily weight or how long it takes you to run 5 miles. Keep a spreadsheet that includes the date, the hour, the measurement, and any other informative variable you think is worth keeping. Do this for 2 weeks. Then make a plot.

[2] https://www.tandfonline.com/doi/abs/10.1080/00031305.2017.1375989