An Organic R Textbook

Ian Cero, PhD MStat

2021

About

This is textbook created during live discussion with the Squeglia Research Group of the Medical University of South Carolina.

Use the **index on the left** to navigate from topic to topic.

Getting Started

Under most circumstances, getting started with R is a straightforward process of downloading and installing a few components. In what follows, we'll talk about what those components are, and the order in which you'll want to install them.

NOTE: the order of installation matters, so please be careful to follow the instructions in the order given below.

Overview of the R ecosystem

Most of the time you are using R for data analysis, you'll want to remember that you are working with a whole ecosystem of analysis tools. Understanding the different roles these tools serve in your project will help you keep track of the best way to use them - and hopefully make your R experience more intuitive.

The **R ecosystem** you'll be using for data analysis generally consists of three parts:

- The **R language**, which is a coding language (like Java or Python) that was optimized for talking to computers about statistical problems. When you download and install "R" (step 1, below), you are teaching your computer how to "speak" that R language.
- The **RStudio Integrated Development Environment (IDE)** is a program that you will use to make it easier to talk to your computer in the R language. Think of R as a language and RStudio as a chat app that has a bunch of features (e.g., your contacts list, spell check) that make the chat experience faster and easier for you.
- **R Packages** are collections of code that other people have written to make R perform particular tasks, usually around a them. For example, there are packages for making R perform new types of analyses, but also for streamlining data cleaning. You can download these packages with R's `install.packages()` command, so that your computer can use them

too. Think of packages like special tricks you are teaching your computer. Once it learns the trick (i.e., installs the package) it can do that new trick with R over and over again, making youre live a lot easier.

Installation

Step 1 - Download and install the R language

The first step to a functioning R ecosystem on your computer is to install the R language on your computer. It's freely available at the Comprehensive R Archive Network (CRAN), which is an acronym you'll see a lot as we go forward. CRAN is just a group of programmers in charge of maintaining and updating the R language.

To install R, go to `https://cran.r-project.org/`. Then at the very top of the page, choose the installer that is right for your operating system (i.e., Windows, macOS, Linux).

HINT: Depending on your operating system, the downloads page can be kind of intimidating. What you are looking for is the most updated version of R, which as of today (2021-12-01) is R 4.1.2. If you find that you want something to take you through the process at a more step-by-step pace, this tutorial (`https://www.datacamp.com/community/tutorials/installing-R-windows-mac-ubuntu`) should have an answer for each operating system.

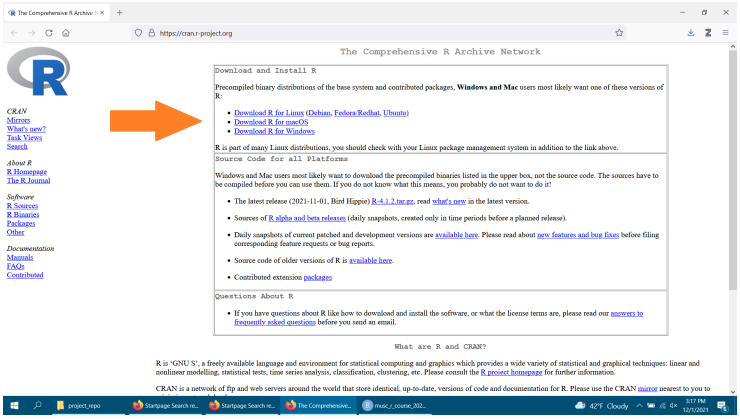

Figure 1: https://cran.r-project.org/

Step 2 - Download and install the RStudio IDE

Several years ago, writing code in R was especially difficult because there was so much to keep track of and it was all hidden behind the code. The RStudio IDE fixed that for us by allowing us to continue coding in R, but this time with a collection of useful windows that keep track of what's happening in our code (e.g., what datasets do we have loaded? what plots have we generated?).

After you installed R, installing the RStudio IDE should be fairly straightforward. Just go to their Downloads page (`https://www.rstudio.com/products/rstudio/download/`) and choose the **Desktop Version**

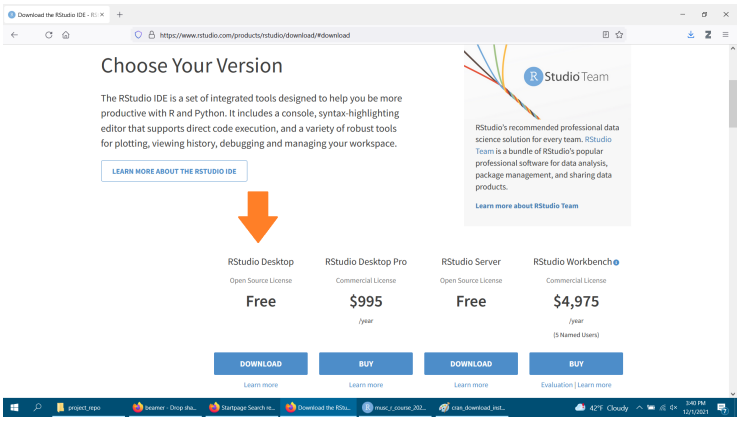

Figure 2: https://www.rstudio.com/products/rstudio/download/

NOTE: make sure you finish step 1 first! This will allow you to save several steps linking R and RStudio. This is because if R is installed first, RStudio will do the linking for you automatically.

Step 3 - Install the `tidyverse` package (optional)

Now that both R and RStudio are installed, let's open RStudio and install some packages.

1. Once you have Rstudio open, you should see several windows. Find the Console window.
2. Inside that window, type `install.packages('tidyverse')` and press ENTER.
    - R is case-sensitive, so make sure to type (or copy/paste) the command exactly.

- This should start an installation process that takes a few minutes (no more than 10) and will install a package you will use basically every time you program in R - so it's very useful to have.
- If you get an error message while installing, don't worry! That's pretty common and you've probably still done everything right. Just remind me in class and we will make sure to troubleshoot it for you.

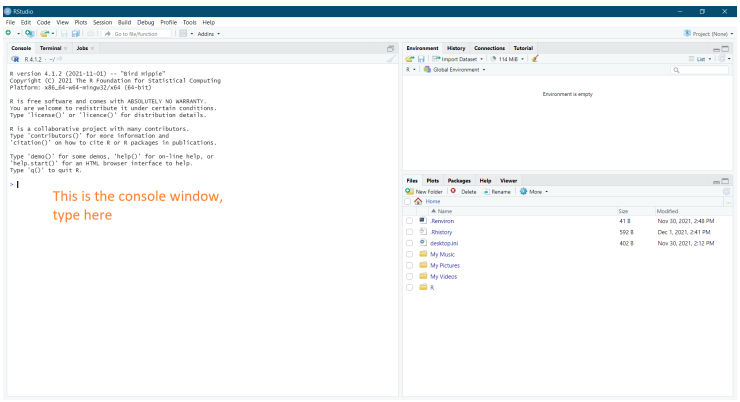

Figure 3: The RStudio IDE

A tour of RStudio

If R is a language, RStudio is a chat program that makes it easier to talk to your computer using that language. It includes multiple windows that help you keep track of the different parts of the conversation.

Although there are lots of tabs scattered throughout the overal RStudio application, there are generally 3 that we will use every day.

The console

Shown in the left half (or sometimes lower left quarter) of the screen. The console is where you can talk to R live. Everything you enter into the console happens right away, which makes it really useful for quick calculations.

The environment

The Environment tab in the top right quadrant shows you every object you currently have imported into R. This is especially useful for keeping track of

what you named your datasets (and whether your datasets even made it into R in the first place).

The lower right pane

There are many tabs in the lower right pane and you'll use most of them on a daily basis. The files tab shows you all the files in your current working directory (the file that R is paying attention to right now). The plots pane shows your plots, assuming you haven't told R to send them somewhere else. Lastly, the help pane will show you R's (very useful) help documentation, anytime you put a `?` in front of a command (e.g., `?lm()` brings up the help file for the `lm()` command).

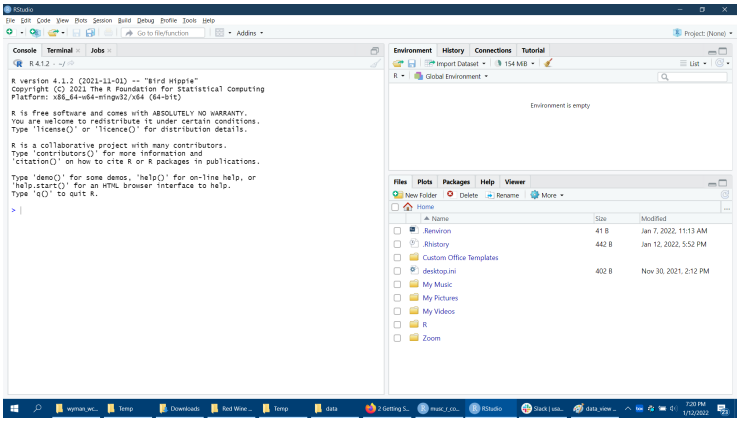

Figure 4: The RStudio IDE

Your very first analysis

To give us a roadmap for our future work in R, we'll start with a basic analysis here. For demonstration purposes, we'll be doing a basic analysis of red wine and checking whether its chemical properties predict how well it's rated by professional tasters.

Step 1 - download the data

The first step is simply to download the data, **which can be downloaded here.**

You might be asked to create a Kaggle account or to log in with Google. Don't worry though, it's totally free.

Step 2 - Make an RStudio Project

Now that we have our data, we need a place to store it - along with all the other important things we'll be working on, like our code and analysis output. The best option is to create an RStudio Project, which is a special kind of folder that RStudio knows to keep track of. RStudio projects have a number of advantages, but for now all you need to know is that they make it easier to keep track of your data.

To make a project...

1. Navigate to File » New Project (sometimes this takes a few seconds to load after you click on it)
2. Select New Directory
3. Select New Project
4. In the Directory Name text box, write the name for your project. In this case, a good name might be something like "Red Wine Practice."
    - Note, you can change the directory you want your project folder too, but it's not necessary for this example.
    - Leave all the remaining boxes (Create git repository, Use renv with this project) **unchecked**.
5. Click Create Project

With your project now created, you should now see "Red Wine Practice" (or whatever you named your project in the top of your RStudio application window). Moreover, if you look to the Files pane on the lower right, you should see a file called `Red Wine Practice.Rproj`. Lastly, you should notice that your working directory is now called "Red Wine Practice."

You can double-check this by typing `getwd()` (short for "get working directory") into the R Console on the bottom left and hit ENTER.

Step 3 - Get the data into your project folder

The quickest way to get your data into your project folder, is simply to copy/paste the `winequality-red.csv` you downloaded in Step 1 into your Red Wine Practice folder.

Where is that practice folder? Again, you can get the full path for your project folder simply by typing `getwd()` into the R console on the lower left and hitting ENTER.

To check whether your copy/paste operation worked, your can type `list.files()` into the R console. If it worked, you should see it listed along with your `Red Wine Practice.Rproj`

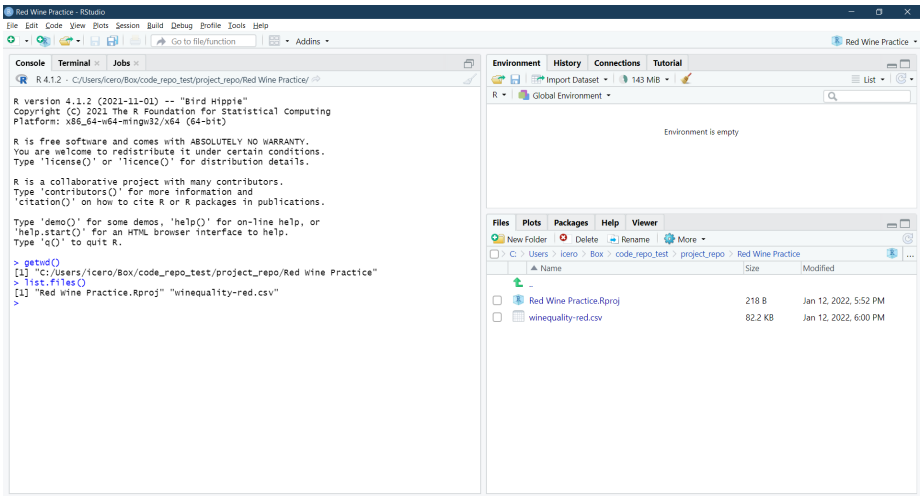

Figure 5: Checking the copy/paste operation worked with ‘list.files()‘

Step 4 - Open an Rmarkdown Notebook

We need a place to type our R commands, plus some notes to ourselves. The best way to do both of those things at the same time as an Rmarkdown notebook.

1. Navigate to File » New file » R Notebook
    - **NOTE**: You might get a message asking if you want to install some packages. Press OK / Yes - you *do* want to install them.
2. With the new notebook file opened, press CTRL+S to save the file under a different name. You can use whatever name you want. For this example, I will use `main.Rmd` to remind myself this is my main analysis file.
3. Inside your newly saved file, change the title from “R Notebook” to something more descriptive like “My first wine analysis.”
4. Lastly, RStudio gave us a bunch of boilerplate code. We won’t need that today, so delete everything below the second `---` at the top of the page, right under `output: html_notebook`. Your final document should then look something like the following.

Step 5 - Create a space to code

Rmarkdown documents have whitespace and greyspace. Whitespace is where you type notes to yourself. Greyspace is where you type R commands. We call these greyspaces **code blocks**.

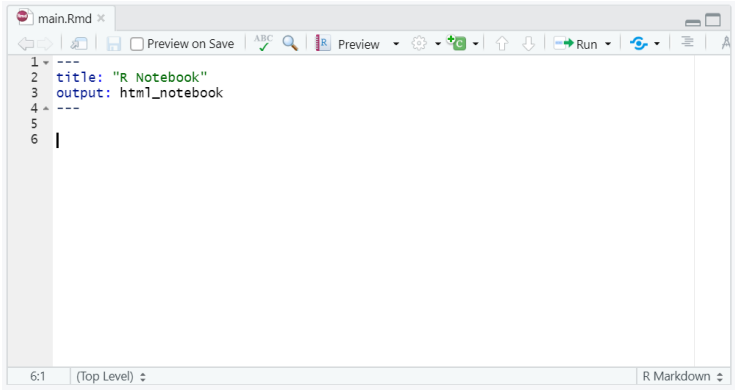

Figure 6: What your Rmarkdown file should look like before we start coding

1. Anywhere below line 4, type a note to yourself like "This is where I imported the data."
2. Move your curser to a line below that note you just wrote (e.g., line 8). Then, press CTRL+ALT+I. This will create a grey codeblock.

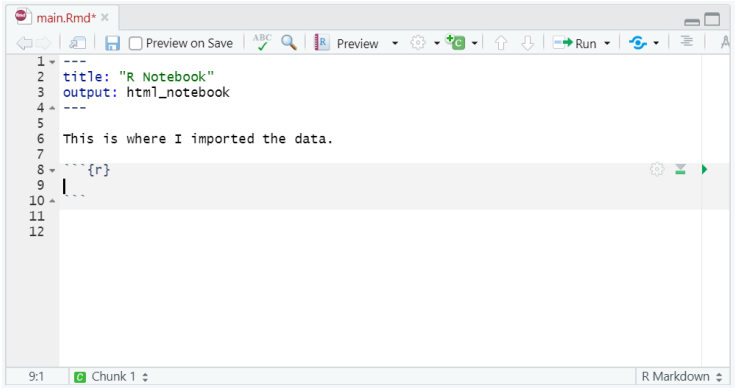

Figure 7: Adding a note to yourself and a grey code block

Step 6 - Write a command to import the data

Importing data involves four things: the name of the datafile, a command to read the data into R, the "assignment operator" (written as `<-`), and the name you want R to call the imported data when you reference it later. Fortunately, this *sounds* much more complicated than it is.

For now, just copy and paste the following commmand into the middle of your grey code block (for me, that is line 9).

```r
my_data <- read.csv('winequality-red.csv')
```

This is R code. It can be translated into English, but it's a little clunky. Also, generally code works from right to left. So, you would read this sentence as something like "Take the file called `winequality-red.csv`, read it into R, then call whatever comes out of that process `my_data`."

To get the R code to run, put your cursor inside the grey code block and press CTRL+SHIFT+ENTER. This runs all the code inside that block.

Step 7 - Look inside the data

Make a new code block a few lines down from the last one. Then type just the name of your imported data and run the block. You should see a preview of your dataset. To scroll through the other variables in the dataset, press the right-facing triangle on the right side of the preview.

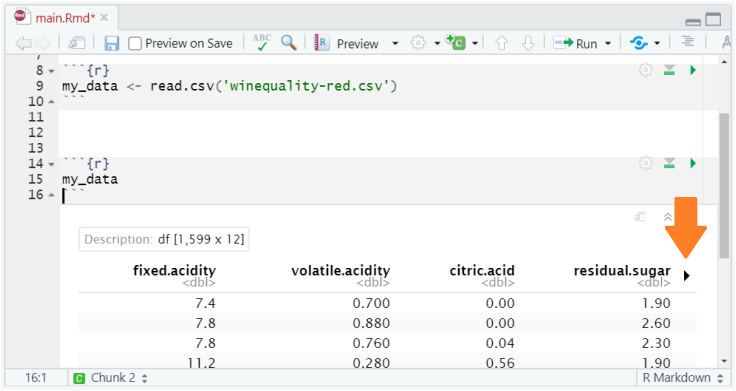

Figure 8: Preview of the data

Step 8 - Make a histogram

I think I'm most interested in the final variable in the dataset, `quality`. In this case, that is the quality of the wine - rated by professionals on a scale of 1 to 10. Let's see what the distribution looks like. For that we can use the `hist()` command (short for "histogram").

But what should we give our `hist()` command? We unfortunately can't give it the whole dataset. After all, we only want a histogram of one variable. How do we specify that variable? We use the "selection operator," which we write as a `$`-symbol, like below.

```
hist(my_data$quality)
```

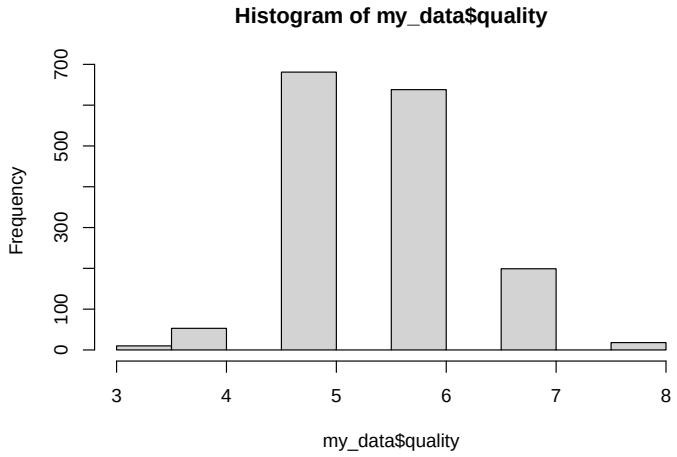

Figure 9: Histogram of wine quality ratings

Step 9 - Run a regression

Now that we have a sense of what our outcome variable (`quality`) looks like, let's see if we can investigate some of the chemical characteristics in wine associated with that outcome.

The ones that stick out to me are `pH` (acidity) and `alcohol` content because they seem like things that would really affect the taste. Let's use those as our predictors

To run a regression, we need just a few things (out of order): - The `lm()` command, which is short for "linear model." This is how R will know we want a regression. - Our data, named `my_data` - A regression formula, which tells R what the outcome variable and it's predictors are - The assignment operator again (`<-`), which tells us where to store the results - A name for where to store the results

Putting all of that together looks like this. One you have it all typed in (or copy/pasted), run the whole block with CTRL+SHIFT+ENTER.

```
my_results <- lm(
  formula = quality ~ pH + alcohol,
  data = my_data)
```

Step 10 - Get a summary of your results

You may have noticed that in Step 9, your results didn't show up anywhere after you ran your regression. That's because R stored them in `my_results`, just like it stored the outcome of the `read.csv()` command in `my_data`.

This often surprises people who come from other software packages, but that's okay. Our results are still easy to get. We just need to ask R for a summary of them.

```
summary(my_results)
```

```
##
## Call:
## lm(formula = quality ~ pH + alcohol, data = my_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.7153 -0.4066 -0.1105  0.5076  2.4584
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.42581    0.38742  11.424  < 2e-16 ***
## pH          -0.85011    0.11571  -7.347 3.23e-13 ***
## alcohol      0.38617    0.01676  23.036  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6989 on 1596 degrees of freedom
## Multiple R-squared:  0.252,  Adjusted R-squared:  0.2511
## F-statistic: 268.9 on 2 and 1596 DF,  p-value: < 2.2e-16
```

This gives us a basic regression table with all of the same information you are used to. Under the *Coefficients* heading, we see the b-values / slopes (Estimate column), their standard errors, t-values, and p-values.

Next to the p-values, we see stars reminding us that our p-values are significant at the $< .001$ level. In fact, our p-values are so small, they have to be shown in scientific notation ("3.23e-13"). These can be treated as basically zero.

Lastly, at the very bottom, we also see the usual R-squared values (here, .252), our F-statistic, and degrees of freedom - everything we need to create a publication-ready regression table.

A Look forward

Over the last 10 steps, we ran through a basic version of essentially every R analysis you are likely to conduct in the future. This is example thus contains a useful workflow you will likely want to recreate in your future work:

- Create a project to store everything
- Import the data
- Visualize / explore the data
- Run main analysis model
- Summarize results

Although the data and models you use might be more complicated as your time in R progresses, it's helpful to remember that everything your are doing typically reduces to these fundamental steps.

Basic R

Writing in R and Rmarkdown

Chatting with R

Using R is just a chat with the computer.

"Hey, R. What is $1 + 2$?"

```
1 + 2
```

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

Rmarkdown tricks

- To make text **bold**, we add two **s around it.
- To make text *italicized*, we add just one * around it.
- If we need special characters (like * or $), then we just add a forward "\" in front of them (but not behind).
- Math symbols in your text are process with Latex, just put an "$" before and after your math. Like this, $y = x$ becomes $y = x$.

Code blocks

To make a code block, press CTRL+ALT+I.

```
banana <- 5
banana + 1
```

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

Variables

Variables are values that I want to give names to and save for later.

The assignment operator

We make variables with the <- operator. This is called the *assignment operator* because it assigns values on the right to names on the left. If I want to know what the value of a variable is, I can run it alone on its own line.

```
my_special_var <- 1 + 2
my_special_var
```

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

You can TECHNICALLY use = for assignment too. Never do this.

```
my_other_var = 12
my_other_var + my_special_var
```

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

The = symbol gets also used for a few other things in R. So, using it to assign variables will make your code more confusing to you, when you go back to read it over later.

Numerics

Doubles

Doubles are decimal numbers, like $1.1, 2.2, 3.0$. If I make a number variable without doing anything special, R defaults to a double.

```
a <- 1.1
b <- 2.0
is.double(a)
```

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

```
is.double(b)
```

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

Integers

Integers must have an `L` after them. That is how R knows that you don't want a double, but instead want a "long-capable integer."

```
c <- 1L
d <- 1
is.integer(c)
```

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

```
is.integer(d)
```

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

Here is a useful cheatsheet for the different numeric operators and how they behave.

Operator	Expression	Result
+	10 + 3	13
-	10 - 3	7
*	10 * 3	30
/	10 / 3	3.333
^	10 ^ 3	1000
%/%	10 %/% 3	3
%%	10 %% 3	1

Why care about the difference?

Almost 99% of the time, this wont matter. But, with big data, integers take up must less memory.

```
my_integers <- seq(from = 1L, to = 1e6L, by = 1L)
my_doubles <- seq(from = 1.0, to = 1e6, by = 1.0)
object.size(my_integers)
```

```
## 4000048 bytes
```

```
object.size(my_doubles)
```

```
## 8000048 bytes
```

Note here that although we are using only whole numbers from 1 to 1 million, the first sequence (`my_integers`) is stored as an integer and the second sequence (`my_doubles`) is stored as a number that may include decimals. This second case needs more space (twice as much) to be allocated in advance, even if we never use those decimal places.

Again, this will almost never matter for most people, most of the time. However, it is good to be aware of for when your datasets get large (i.e., several million cases or more).

Characters

Characters are text symbols and they are made with either "" or '', either works.

```
a <- 'here is someone\'s text'
b <- "here is more text"
a
```

```
## [1] "here is someone's text"
```

```
b
```

```
## [1] "here is more text"
```

To combine two strings, I use `paste()`.

```
paste(a, b)
```

```
## [1] "here is someone's text here is more text"
```

If I dont want a space, then I used `paste0()`.

```
paste0(a, b)
```

```
## [1] "here is someone's texthere is more text"
```

Booleans

These are True and False values. You make them with the symbols `T` or `TRUE` and `F` or `FALSE`.

```
x <- T
y <- F
```

To compare them, we can use three operators.

- `&` is "and"
- `|` is "or"
- `!` is "not" (just give me the opposite of whatever is after me)

```
x & y # false
```

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

```
x | y # true
```

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

```
x & !y # true
```

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

We can also have nested equations

```
z <- F
x & !(y | z) # true
```

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

We can also compare numbers.

```
a <- 1
b <- 2
```

```
a < 1
```

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

```
a <= 1
```

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

```
a == 1
```

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

If I want to compare multiple numbers, I need to do it seperately.

```
(a > 1) | (b > 1)
```

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

Remember that booleans are ultimately numeric values underneath.

```
d <- T
k <- F
u <- 5
d*u
```

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

```
d*k
```

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

```
as.numeric(d)
```

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

```
as.numeric(k)
```

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

Special types

`NA` - missing

```
is.na(NA)
```

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

`NaN` - you did math wrong

```
0/0
```

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

`Inf` - infinity

```
-5/0
```

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

Vectors

R is built is on vectors. Vectors are collections of a bunch of values of the same type.

```
my_vec <- c(1, 5, 3, 7)
my_vec
```

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

If I try to put different types together, they go to the most primitive type (usually a character string).

```
my_other_vec <- c(22, 'orange', T)
my_other_vec
```

```
## [1] "22"     "orange" "TRUE"
```

```
my_third_vec <- c(T, F, 35)
my_third_vec
```

```
## [1]  1  0 35
```

We can also missing values.

```
my_fourth_vec <- c(1, 4, 5, NA)
my_fourth_vec
```

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

```
is.na(my_fourth_vec)
```

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

If I want to combine two vectors...

```
a <- c(1, 2, 3)
b <- c(3, 5, 7)
c(a, b)
```

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

A brief example of matrices

```
matrix(
  data = c(a, b),
  nrow = 2,
  byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    5    7
```

Sometimes I want special vectors, direct sequences of numbers. There are two ways to do this. If all I want is a integer sequence (made of doubles), then I use the "`<first number>:<last number>`."

```
1:5
```

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

```
5:1
```

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

Other times, I need to count by something other than one, so I use `seq(from = <start>, to = <end>, by = <number to count by>)`

```
seq(from = 1, to = 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

Hint: for brevity, I can leave off function parameter names, **as long as I enter them in order**

```
seq(1, 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

If I add a constant to a vector, then they all go up by that constant.

```
1:5 / 3
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667
```

I can do math with equal-length sequences too.

```
1:5 - seq(1, 4, by = .7)
```

```
## [1] 0.0 0.3 0.6 0.9 1.2
```

But they **must** be equal lengths.

```
1:5 / 1:4
```

```
## Warning in 1:5/1:4: longer object length is not a multiple of shorter object length
```

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

To access the elements of a vector, I put a number OR booleans in brackets `[]`.

```
my_vec <- c('apple', 'orange', 'banana', 'pair')
my_vec[2]
```

```
## [1] "orange"
```

```
my_vec[2:4]
```

```
## [1] "orange" "banana" "pair"
```

```
my_vec[c(3, 2, 1, 4)]
```

```
## [1] "banana" "orange" "apple"  "pair"
```

I can also use bools.

```
my_other_vec <- c(1, 4, 6, 7, 9, 3, 9)
my_other_vec < 5
```

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

```
my_other_vec[my_other_vec < 5]
```

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

I can also use functions that return values to access vectors, if I am creative...

```
my_other_vec[max(my_other_vec) == my_other_vec]
```

```
## [1] 9 9
```

R also has special vectors that are pre-loaded. The most commonly used are `letters` and `LETTERS`, which return the lower-case letters and uppercase letters of the English alphabet, respectively.

```
vec <- c(1, 3, 4, 5, 3, 2, NA)
mean(vec, na.rm = T)
```

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

Lists

« More on lists to come »

Lists are special vectors that can hold multiple types of elements, even vectors

```
my_vec <- c(4, 5, 6)
my_list <- list(1, 'banana', 3, NA, my_vec)
my_list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "banana"
##
```

```
## [[3]]
## [1] 3
##
## [[4]]
## [1] NA
##
## [[5]]
## [1] 4 5 6
```

Dataframes

Construction

Dataframes are spreadsheets. Under the hood of R, they are just lists of vectors, where all the vectors are required to be the same length. To make one, you can call the `data.frame()` function and put your vectors inside.

```
heights <- c(60, 65, 71, 72, 64)
sexes <- c('female', 'female', 'male', 'male', 'female')
shoes <- c('Adidas', 'Nike', 'Nike', 'Salvatore Ferragamo', 'Reebok')
df <- data.frame(height = heights, sex = sexes, shoes = shoes)
df
```

```
##   height    sex               shoes
## 1     60 female              Adidas
## 2     65 female                Nike
## 3     71   male                Nike
## 4     72   male Salvatore Ferragamo
## 5     64 female              Reebok
```

Built-in dataframes

R has numerous built-in datasets that are ideal for demonstration purposes. We can get access to them using the `data()` command. This will load the data into our session, so we can then look at it.

```
data('mtcars')
mtcars
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
```

```
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Some datasets do not come in the form of a dataframe right away, but they can be converted into one using the `as.data.frame()` function.

```
data(Seatbelts)
is.data.frame(Seatbelts)
```

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

```
seatbelts_df <- as.data.frame(Seatbelts)
is.data.frame(seatbelts_df)
```

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

Functions

A function is a piece of code that does work for you. It takes inputs and (usually) returns outputs. For example, the `sum()` function takes the sum of a numeric vector.

```
my_vec <- c(3, 6, 2, 3)
sum(my_vec)
```

```
## [1] 14
```

Getting help

If I ever need to know something about a function, I can put a question mark in front of it (no `()`s) and run that line. That will bring up the help document for that function.

```
?sum
```

Function parameters

In addition to the data they take as input, most functions have additional *parameters* (sometimes called "arguments," but they mean the same thing). Looking at its help file, the `sum()` function has two parameters:

- `...`, the numbers you want to sum
- `na.rm = FALSE`, which tells `sum()` whether you want to remove ('rm') missing values ('na') before summing.

Let's look at what happens when we try to `sum()` a vector with a missing value.

```
my_vec <- c(5, NA, 2, 3) # should be 10
sum(my_vec)
```

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

R tells us the answer is missing (`NA`) because at least one of the vector elements is missing. This is to be conservative and to force you never to ignore missing values by accident. But what do we do if we really do want to sum all available values, ignoring the missing values.

Again, looking at the help file, we can see that the `na.rm` parameter of the function is followed by `= FALSE`, under the **Usage** heading of that help document (look for `sum(..., na.rm = FALSE)`). This tells us that the parameter `na.rm`, which tells `sum()` whether to remove missing values from the calulation, defaults to `FALSE`.

To get `sum()` to ignore the missing values in our vector, we simply set `na.rm` to `TRUE` (or `T` for short).

```
sum(my_vec, na.rm = T) # should be 10
```

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

Packages

Packages are collections of functions that someone else put together for you. You can install them using the `install.packages()` function, with the name of your package inside the `()` - don't forget to use either single (`' '`) or double quotes (`" "`) around the package name too.

```
install.packages('ggplot2')
```

Once installed, use the `library()` function to load your package into your R session. Note, you don't need quotes here.

```
library(ggplot2)
```

Error messages

Whenever R detects that something has gone wrong, it will send you an error message in the form of some scary-looking red text.

```
'100' / '2' # trying to divide two strings
```

```
## Error in "100"/"2": non-numeric argument to binary operator
```

Unfortunately, R is not especially smart and is usually bad at detecting exactly WHAT has gone wrong. In this case, all it knows is that `/` needs two numbers - one on either side - to work correctly. It detected something other than that, which is what it told us: there was a non-numeric argument (input) SOMEWHERE on either side of the `/` symbol.

Stumbling on an error and getting stuck is especially common. It's also common to get frustrated when you are stuck with the same error for more than a few minutes. For that reason, if you can't solve an error after a few quick tries, it's best NOT to beat your head against the wall. Instead, go to a place like **https://stackoverflow.com/**, which is a website devoted entirely to answering questions - most of which are about coding errors just like yours.

Simply copy and paste your error into the search box and look for someone who asked your question already. You'll notice that many people have had your same problem and have even produced some code you can copy/paste to fix your current issue.

Coding Conventions

R is a language, much like English or Spanish. It sometimes has rules for how you MUST say something in order for your computer to understand at all. For example, R won't let you add a number and a letter together because that wouldn't make sense mathematically.

```
1 + 'a'
```

```
## Error in 1 + "a": non-numeric argument to binary operator
```

Other times, R will let you do the same thing in more than one way. For example, I can name my variables with a mixture of capitals and lower-case letters.

```
apple <- 123
BANANANA <- 456
ClEmEnTiNe <- 789
```

Some of these options might be more confusing than others, but R will technically let you do them.

Other examples include using `<-` or `=` to assign values to variables. As discussed above, both with work.

```
peas <- 'tasty'
carrots = 'also tasty'
```

What should my code look like?

Whenever there are multiple options for how to code, it's worth thinking about whether one will be better for you than the others. If you come up with a

consistent rule over time - like "never use capital letters in function names" - you've developed some **coding conventions**. These achieve a few things for you, but mostly we develop these informal language rules for clarity.

They make it easier for us to read our code, and for others to understand what we were doing when they look at our code later. Some common coding conventions most R users now employ are given below.

Common coding conventions in R

Never use `=` to assign variable values. Use the `<-` operator instead because it is more clear.

```
a = 2 # bad

a <- 2 # good
```

Avoid using `.` to seperate words in your variable and function names because this makes it hard for people who come from other coding langauages to understand us. Use `_` instead.

```
my.favorite.number <- 3.14159 # bad

my_favorite_number <- 3.14159 # good
```

Whenever possible, stick to lower-case variable names. It will make it easier for you to reference your variables later, without accidentally making a capitalization error.

```
apple <- 123 # good
BANANANA <- 456 # bad
ClEmEnTiNe <- 789 # bad
```

Whenever possible, try to use single-quotes (`'`) for character strings, rather than double quotes (`"`). Single quotes are easier on your eyes when you are looking at a page full of code.

```
peas <- "tasty" # bad
carrots = 'also tasty' # good
```

Official coding conventions

Coding conventions are so important, that many people have tried to publish some. You can think of these like stlye guides that many people agreed to use. The one most relevant to our own work here is Hadley Wickham's guide at **https://style.tidyverse.org/**.

Working with ABCD

In this section, we'll add specific content for the ABCD dataset issues as we uncover them over time.

Importing ABCD datafiles

One of the trickiest parts of working with the ABCD dataset is just getting the data into R. Under most circumstances, getting a file into R is as simple as loading the `tidyverse` and telling the `read_csv()` function where to look for your data.

```
library(tidyverse)

wine_df <- read_csv('data/winequality-red.csv')

head(wine_df)
```

```
## # A tibble: 6 x 12
##   `fixed acidity` `volatile acidity` `citric acid` `residual sugar` chlorides `free sulfur dio
##             <dbl>              <dbl>         <dbl>            <dbl>     <dbl>
## 1             7.4               0.7           0                 1.9     0.076
## 2             7.8               0.88          0                 2.6     0.098
## 3             7.8               0.76          0.04              2.3     0.092
## 4            11.2               0.28          0.56              1.9     0.075
## 5             7.4               0.7           0                 1.9     0.076
## 6             7.4               0.66          0                 1.8     0.075
## # ... with 6 more variables: total sulfur dioxide <dbl>, density <dbl>, pH <dbl>, sulphates <d
## #   alcohol <dbl>, quality <dbl>
```

What's different about ABCD?

There are a few things that are unique about ABCD datasets that make them a little bit harder to import than normal:

1. They are tab-delimited, rather than comma delimited. This means that when they were being saved, the variables in the file were seperated by a tab (specifically, they were separated by the symbol "\t"), rather than a comma (","). In principle, there is nothing wrong with this. It is just a little unusual and so we need to use a special import function to deal with it (specifically, `read_delim()`.

2. The more complex problem is that ABCD datasets have - in an effort to be helpful - have variable names in the first row of data and variable descriptions in the second row. This confuses R, which is expecting the variable names to be in the first row only, then the data to start in the second row.

Walking through ABCD data importation

We'll do this in a few steps. If you're just looking to copy/paste the code, then skip to the end of this section. If you're looking for information about why we are doing what we are doing, then keep reading.

First, notice that we need to use the `read_delim()` command, rather than the usual `read_csv()`. This tells R we are expecting a delimited file. We also tell that function we are expecting the delimeter to be a tab (`delim = '\t'`), which we could figure out by opening the raw data in a basic text editor, like Notepad, and looking at it.

```
abcd_df <- read_delim('data/abcd_screen02.txt', delim = '\t')

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey  src_subject_id  interview_da
##   <chr>         <chr>            <chr>      <chr>       <chr>           <chr>
## 1 collection_id abcd_screen02_id dataset_id The NDAR G~ Subject ID how~ Date on whi
## 2 2573          6579             47156      NDAR_INV0C~ NDAR_INV0CZBUV~ 07/19/2019
## 3 2573          6581             47156      NDAR_INV0D~ NDAR_INV0D4C1R~ 03/10/2019
## 4 2573          6588             47156      NDAR_INV0D~ NDAR_INV0DKWEM~ 11/30/2019
## 5 2573          6594             47156      NDAR_INV0E~ NDAR_INV0E350J~ 12/16/2019
## 6 2573          6609             47156      NDAR_INV0F~ NDAR_INV0FM9MU~ 11/27/2019
## # ... with 34 more variables: sex <chr>, eventname <chr>, scrn2_select_language___1
## #   scrn_braces_v2 <chr>, scrn_future_braces <chr>, scrn_bracesdate_v2 <chr>,
## #   scrn_bracescallback_v2 <chr>, scrn_nr_hair_v2 <chr>, scrn_nr_hair_metal_v2 <chr
## #   scrn_nr_hair_remove <chr>, scrn_nr_hair_date_v2 <chr>, scrn_nr_hair_callback_v2
## #   scrn_eyeliner_v2 <chr>, scrn_nr_piercing_v2 <chr>, scrn_nr_piercing_remove <chr
## #   scrn_nr_piercing_date <chr>, scrn_nr_piercing_callback <chr>, scrn_surgery_v2 <
## #   scrn_height_v2 <chr>, scrn_weight_v2 <chr>, scrn_concerns <chr>, scrn_concerns_
```

We made some progress, but unfortunately, we've got these variable descriptors stuck in the top row of our data now. To get rid of them, we can re-assign the value of `abcd_df` to be a version of itself without it's first row.

```
abcd_df <- abcd_df %>%
  filter(row_number() != 1)

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey    src_subject_id interview_date interv
##   <chr>         <chr>            <chr>      <chr>         <chr>          <chr>          <chr>
## 1 2573          6579             47156      NDAR_INV0CZB~ NDAR_INV0CZBU~ 07/19/2019     136
## 2 2573          6581             47156      NDAR_INV0D4C~ NDAR_INV0D4C1~ 03/10/2019     154
## 3 2573          6588             47156      NDAR_INV0DKW~ NDAR_INV0DKWE~ 11/30/2019     152
## 4 2573          6594             47156      NDAR_INV0E35~ NDAR_INV0E350~ 12/16/2019     137
## 5 2573          6609             47156      NDAR_INV0FM9~ NDAR_INV0FM9M~ 11/27/2019     140
## 6 2573          6630             47156      NDAR_INV0HFH~ NDAR_INV0HFHM~ 11/12/2020     152
## # ... with 34 more variables: sex <chr>, eventname <chr>, scrn2_select_language___1 <chr>,
## #   scrn_braces_v2 <chr>, scrn_future_braces <chr>, scrn_bracesdate_v2 <chr>,
## #   scrn_bracescallback_v2 <chr>, scrn_nr_hair_v2 <chr>, scrn_nr_hair_metal_v2 <chr>,
## #   scrn_nr_hair_remove <chr>, scrn_nr_hair_date_v2 <chr>, scrn_nr_hair_callback_v2 <chr>,
## #   scrn_eyeliner_v2 <chr>, scrn_nr_piercing_v2 <chr>, scrn_nr_piercing_remove <chr>,
## #   scrn_nr_piercing_date <chr>, scrn_nr_piercing_callback <chr>, scrn_surgery_v2 <chr>,
## #   scrn_height_v2 <chr>, scrn_weight_v2 <chr>, scrn_concerns <chr>, scrn_concerns_desc <chr>,
```

That helped, but R still thinks that all of our variables are character strings. We need to tell R that some of our variables might be numbers and that we want it to guess which ones those are. We can do that with the very handy `type_convert()` function.

```
abcd_df <- type_convert(abcd_df)

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey    src_subject_id interview_date interv
##           <dbl>            <dbl>      <dbl> <chr>         <chr>          <chr>
## 1          2573             6579      47156 NDAR_INV0CZB~ NDAR_INV0CZBU~ 07/19/2019
## 2          2573             6581      47156 NDAR_INV0D4C~ NDAR_INV0D4C1~ 03/10/2019
## 3          2573             6588      47156 NDAR_INV0DKW~ NDAR_INV0DKWE~ 11/30/2019
## 4          2573             6594      47156 NDAR_INV0E35~ NDAR_INV0E350~ 12/16/2019
## 5          2573             6609      47156 NDAR_INV0FM9~ NDAR_INV0FM9M~ 11/27/2019
## 6          2573             6630      47156 NDAR_INV0HFH~ NDAR_INV0HFHM~ 11/12/2020
```

```
## # ... with 34 more variables: sex <chr>, eventname <chr>, scrn2_select_language___1
## #   scrn_braces_v2 <dbl>, scrn_future_braces <dbl>, scrn_bracesdate_v2 <chr>,
## #   scrn_bracescallback_v2 <dbl>, scrn_nr_hair_v2 <dbl>, scrn_nr_hair_metal_v2 <dbl
## #   scrn_nr_hair_remove <dbl>, scrn_nr_hair_date_v2 <lgl>, scrn_nr_hair_callback_v2
## #   scrn_eyeliner_v2 <dbl>, scrn_nr_piercing_v2 <dbl>, scrn_nr_piercing_remove <dbl
## #   scrn_nr_piercing_date <lgl>, scrn_nr_piercing_callback <dbl>, scrn_surgery_v2 <
## #   scrn_height_v2 <dbl>, scrn_weight_v2 <dbl>, scrn_concerns <dbl>, scrn_concerns_
```

Copy/paste-able code

Putting it all together, we can import an ABCD dataset like so.

```
abcd_df <- read_delim('data/abcd_screen02.txt', delim = '\t') %>%
  filter(row_number() != 1) %>%
  type_convert()
```