Lab 1: Introduction to R and RStudio

Table of Contents

You can download the lab [here](lab-1.docx).

# Purpose

The purpose of today’s lab is to start building and strengthening foundational coding skills in R. In labs, we take a functional and active approach to learning R. We believe that the easiest way to *learn* R is by *using* R. Giving you some building blocks and suggesting some strategies for overcoming common coding obstacles will allow you to begin exploring the language. In lab, you never need to actively memorize code chunks or functions. You will become proficient naturally with many hours of practice. Rather, the goal of lab is to expose you to what R can do so that you know what tools you have at your disposal when you are later working through a problem.

Today’s lab will cover:

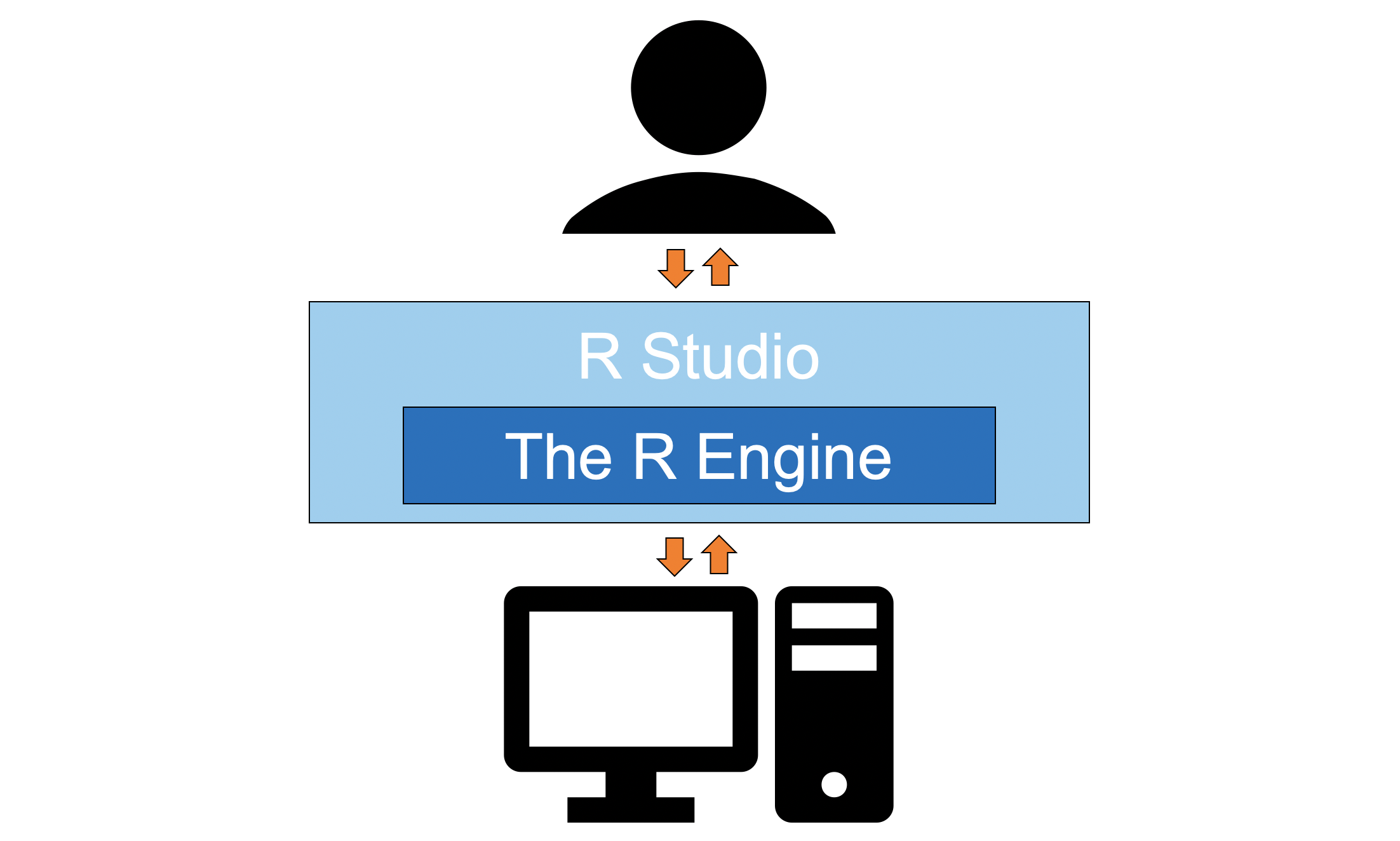
1. [How to download and install R and RStudio](#gs)
2. [The panes of RStudio](#rstudio)
3. [How to create and use R Markdown Documents](#rmarkdown)
4. [Arithmetic commands](#ac)
5. [(Some of) the different types of variables in R](#variables)
6. [What functions are and how to use them](#functions)
7. [How to install and load packages](#packages) and…
8. [How to import data into R](#data)

After we have covered the content of the lab, we will move on to [Minihacks](#minihacks). Minihacks are small coding exercises intended to test your knowledge of the day’s material. The minihacks will be similar to—but narrower in focus than—the questions on your homework assignments. If you are able to successfully complete all of the minihacks, you should be well equipped to begin tackling your homework!

# Getting Started

So what is R?

In the simplest possible terms, R is a programming language used for conducting analyses and producing graphics. It is substantially more flexible than GUI-based statistics programs (e.g., SPSS, LISREL) but less flexible than other programming languages. This lack of flexibility is on purpose; it allows the code to be written in a far more efficient and intuitive way than other programming languages.



Only one piece of software is required to get started using the R programming language and, confusingly, it is also called R. I will refer to it here as the *R Engine*. The R Engine essentially allows the computer to understand the R programming language, turning your lines of text into computer operations. Unlike other popular statistics programs (e.g., SPSS, SAS), the R Engine is free. [Instructions for downloading the R Engine](#download_r) are below.

A second piece of software that is not required to use R but is nonetheless useful is [RStudio](https://www.rstudio.com). [RStudio](https://www.rstudio.com) is an *integrated development environment* (IDE) or, in potentially overly simplistic terms, a tool that makes interacting with the R Engine easier. [Instructions for downloading RStudio](#download_rstudio) are also below.

## Downloading the R Engine

1. Navigate to the webpage for the [Comprehensive R Archive Network](https://cran.r-project.org) (commonly referred to as CRAN).
2. Under “Download and Install R” click the appropriate link for your operating system. For example, if you are using a Mac, you would click on [Download R for (Mac) OS X](https://cran.r-project.org/bin/macosx/).
3. Click the link for the latest release. As of writing this (September, 2021), the newest version is "4.4.1 "Race for Your Life" released on 2024/06/14" (all version nicknames are references to the [Peanuts comic strip](https://twitter.com/snoopy/status/976452382480531458)). I would click R-4.4.1.pkg to start the download.
4. Once the file is downloaded, click on it to open it. Your operating system should guide you through the rest of the installation process.

*Note.* The same steps are used to update the R Engine: You install a new version and replace the old version in the process.

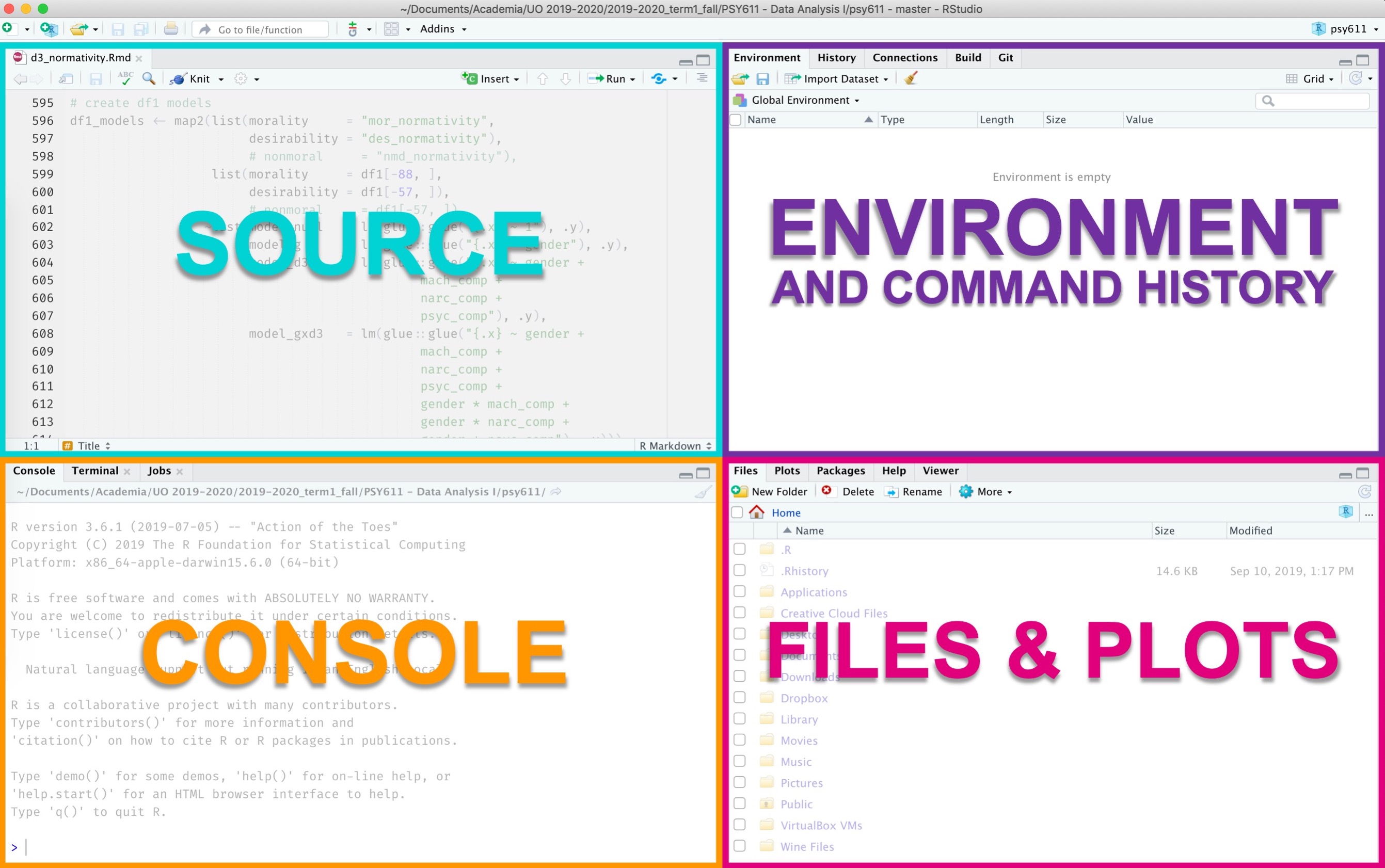
## Downloading RStudio

1. Navigate to the webpage for the free version of [RStudio](https://www.rstudio.com/products/rstudio/download/#download). For our purposes (and for most people’s purposes) the free version of [RStudio](https://www.rstudio.com/products/rstudio/download/#download) is all that you need. The available installers are listed at the bottom of the page under the header “Installers for Supported Platforms.”
2. Select the installer for your operating system. Since I am using a macOS, I would click RStudio-2024.09.0-375.dmg. If you are using Windows 10, you would click RStudio-2024.09.0-375.exe.
3. Once the file is downloaded, click on it to open it. Your operating system should guide you through the rest of the installation process.

*Note.* To update [RStudio](https://www.rstudio.com) after it is already installed, all you have to do is navigate to Help > Check for Updates in the menu bar.

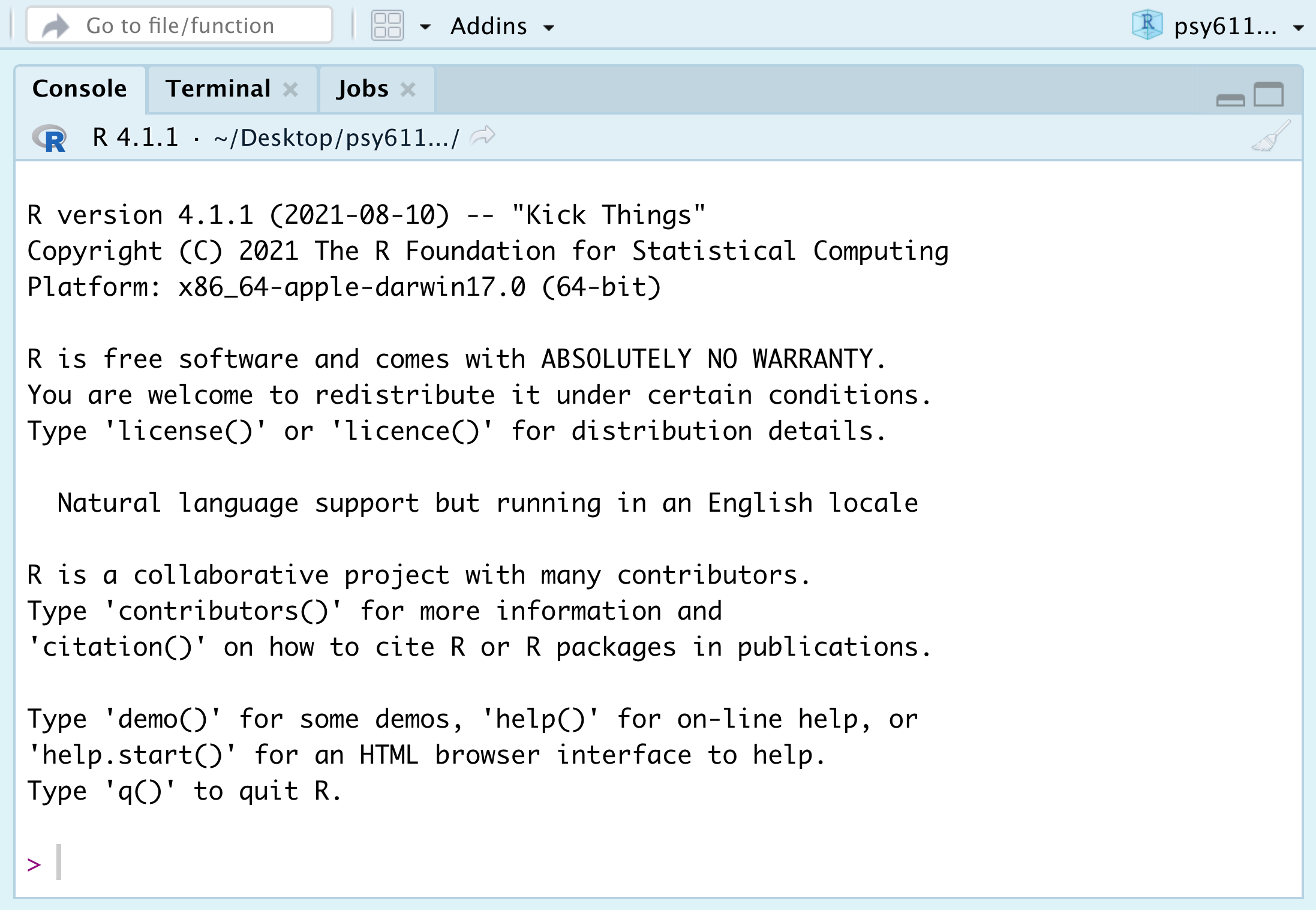
# Features of RStudio

As shown in the image below, an [RStudio](https://www.rstudio.com) session is split into four sections called panes: [the console](#rstudio_console), the [source pane](#rstudio_source), the [environment/history pane](#rstudio_environment), and the succinctly named [files/plots/packages/help pane](#rstudio_files).



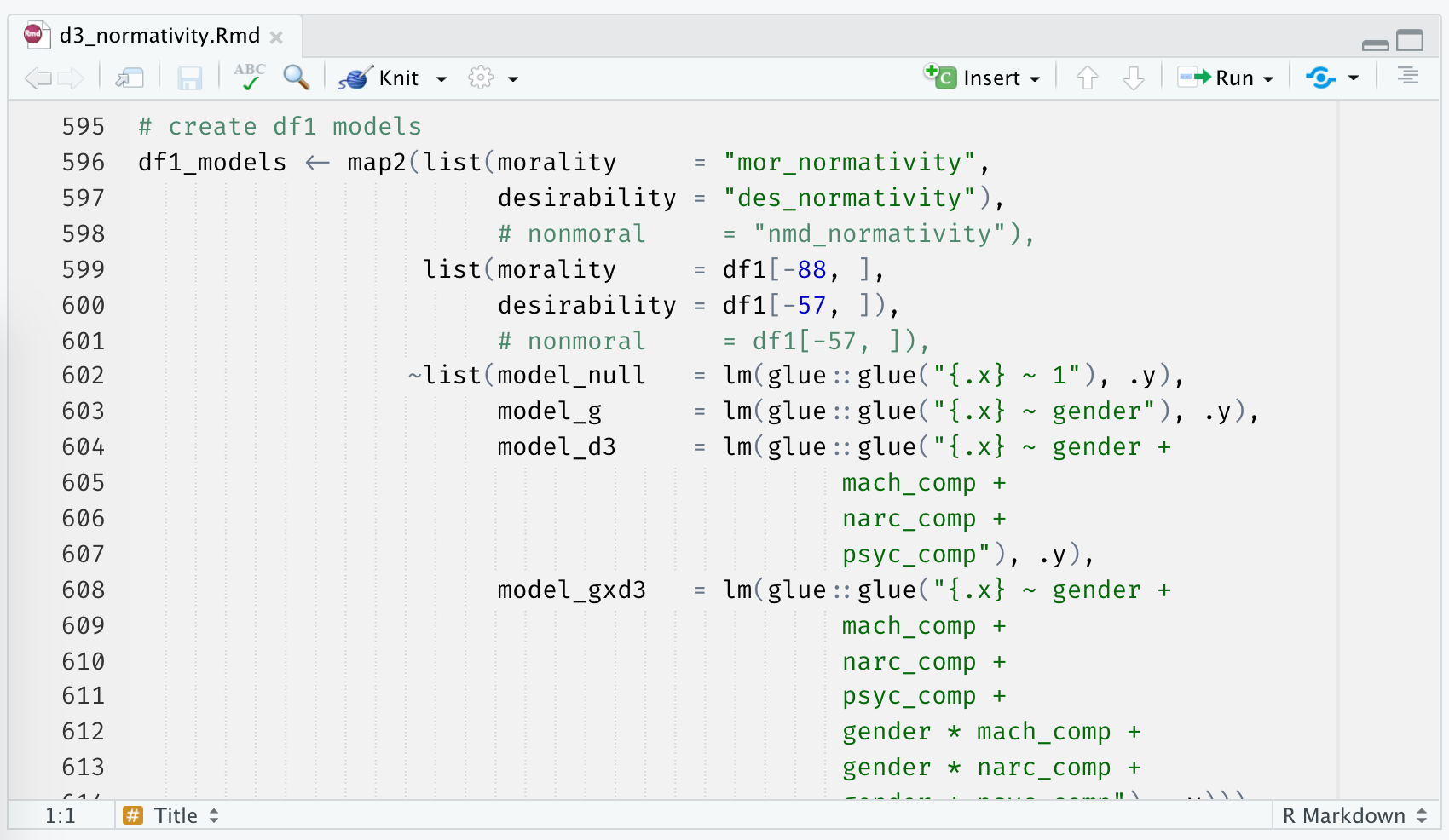
## The Console

In RStudio, the console is the access point to the underlying R Engine. It evaluates the code you provide it, including code called using the [the source pane](#rstudio_source). You can pass commands to the R Engine by typing them in after the >.



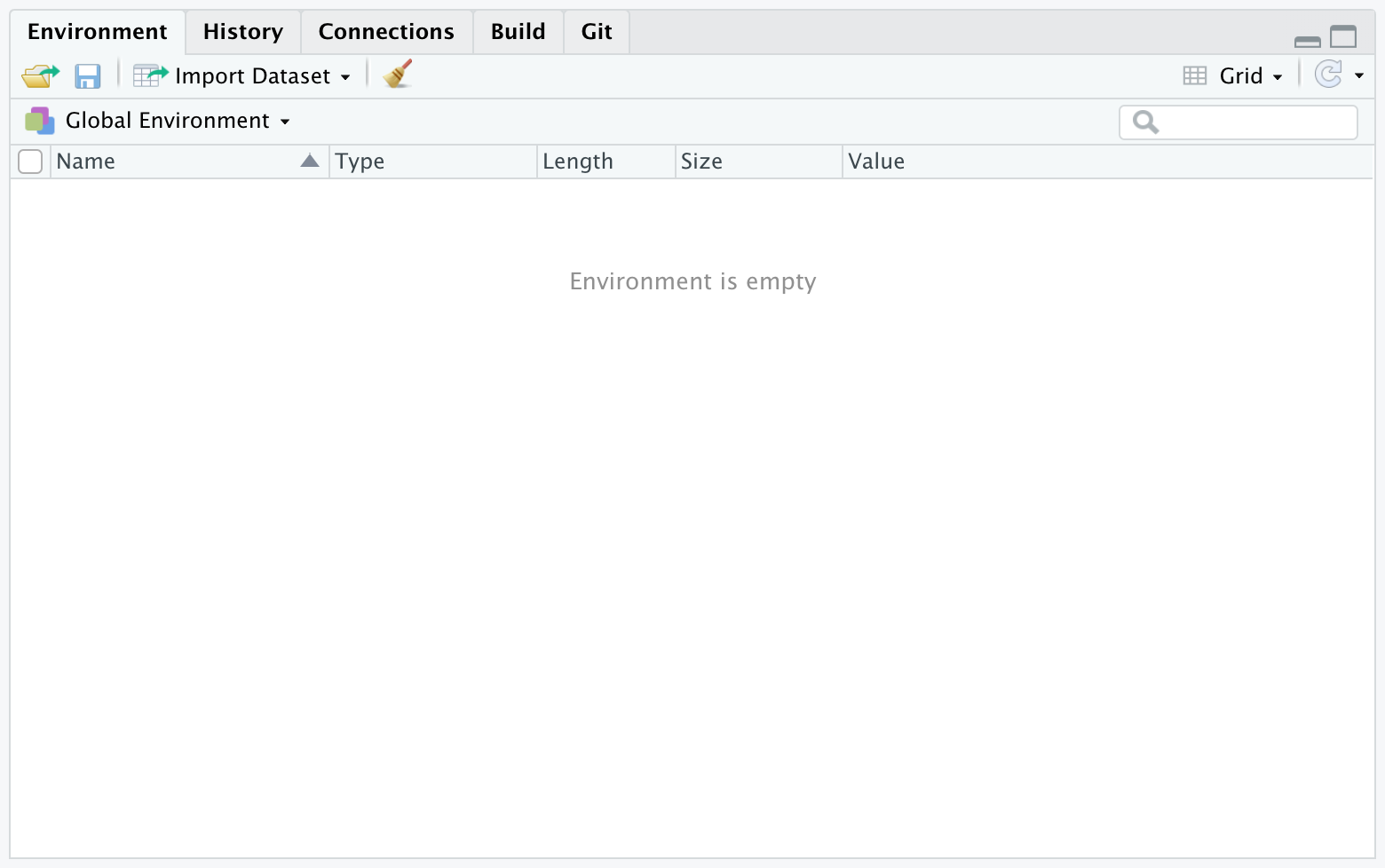
## Source

The source pane shows you a collection of code called a script. In R, we primarily work with R Script files (files ending in .R) or R Markdown documents (files ending in .Rmd). In this class, we will mostly be working with R Markdown files. The document you are currently reading was created with an R Markdown document.



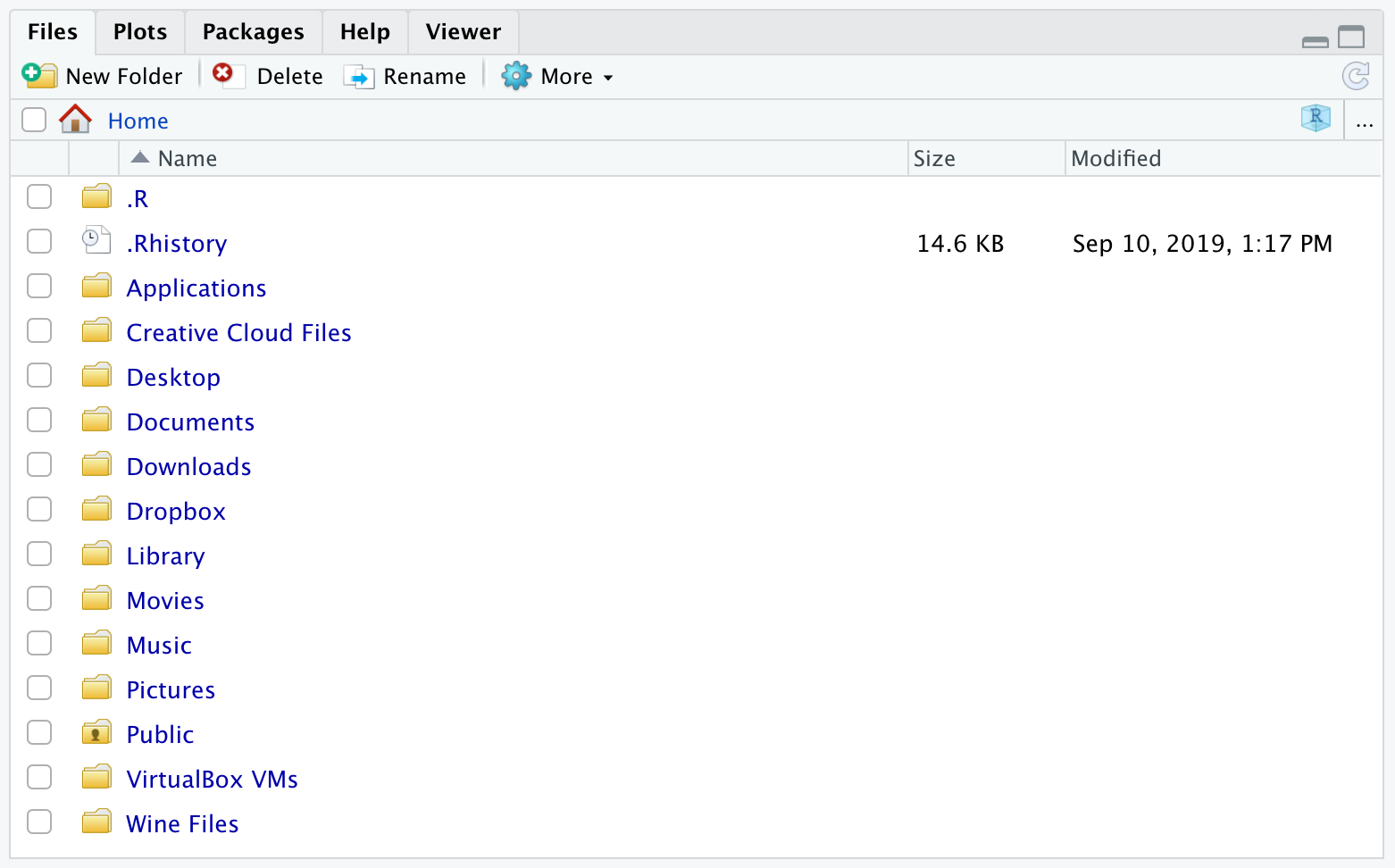
## Environment/History

The environment/history pane shows, well, your environment and history. Specifically, if you have the “Environment” tab selected, you will see a list of all the variables that exist in your global environment. If you have the “History” tab selected, you will see previous commands that were passed to the R Engine.



## Files/Plots/Packages/Help

The final pane—the files/plots/packages/help pane–includes a number of helpful tabs. The “Files” tab shows you the files in your current working directory, the “Plots” tab shows you a preview of any plots you have created, the “Packages” tab shows you a list of the packages currently installed on your computer, and the “Help” tab is where help documentation will appear. We will discuss packages and help documentation later in this lab.



# Projects

Whenever you start a new research project, you should create a new *R Project.* The R project is a working directory where your .RProj file, scripts, data, images, etc. will live. Creating a folder that contains all of the files for your new research project will keep you organized and make it easy for others to download and reproduce your work. We will open up a new project for this class and call it psy611.

## Why Use R Projects?

**1. Organization and Clarity**: R Projects help you keep everything in one place. By having a single folder for all your files, you minimize the risk of losing important pieces of your work. It also makes your workflow more structured since every file related to the project is easily accessible.

**2. Portability**: If you need to work on another machine or share your project with collaborators, R Projects ensure that all paths are relative to your project directory. This means that anyone who has your project folder can immediately run your code without needing to manually adjust file paths.

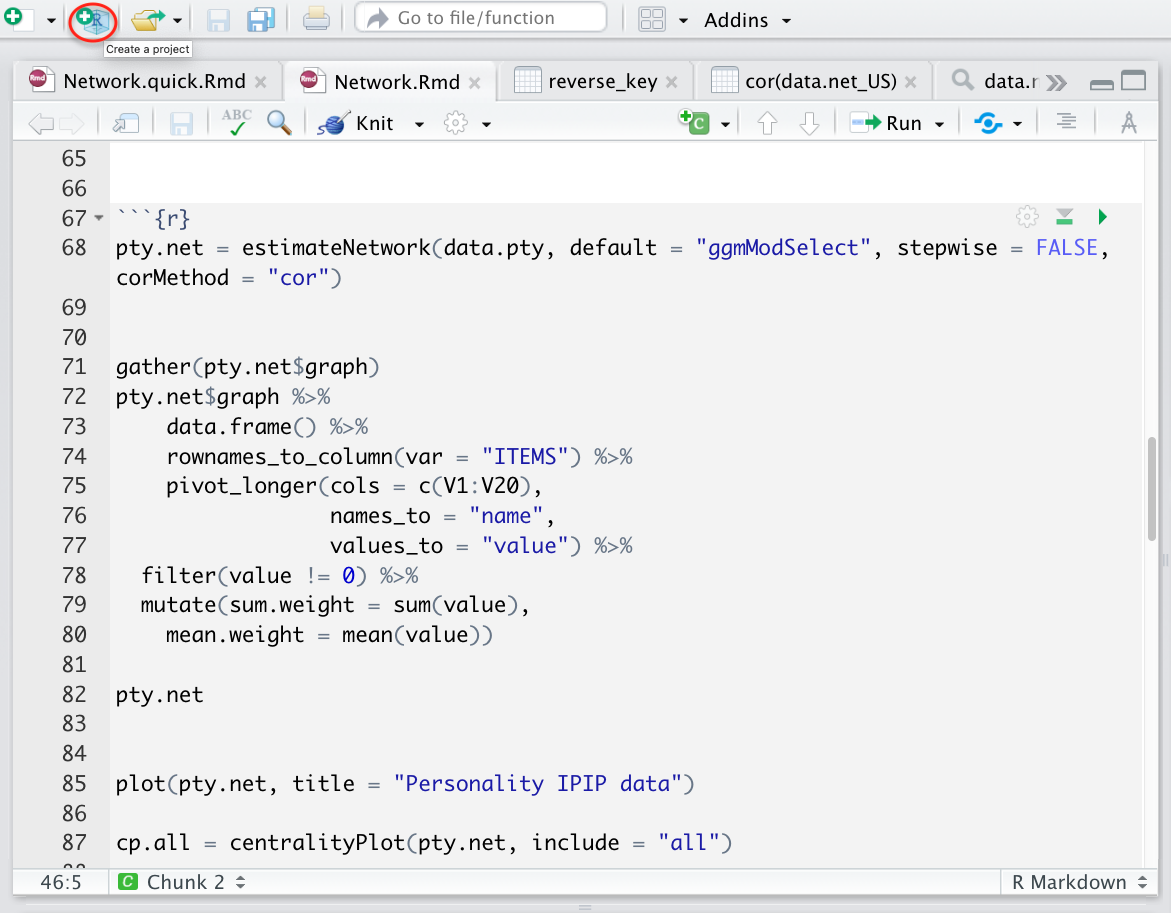
**3. Reproducibility**: Having a dedicated project folder with consistent file organization facilitates reproducibility, which is critical in academic research. Anyone accessing your project can quickly recreate your analyses by running your scripts within the project environment.

**4. Automatic Setting of Working Directory**: When you work within an R Project, RStudio automatically sets your working directory to the project folder, reducing the need to manually set file paths each time you work on your code.

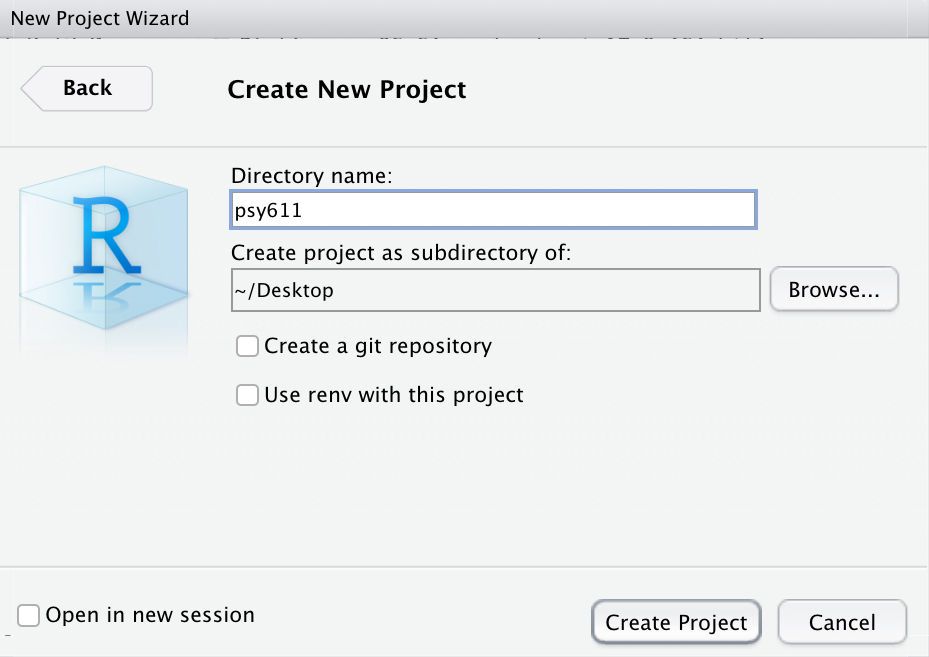
**5. Version Control Compatibility**: If you’re using Git or another version control system, R Projects make it easier to track changes across scripts and ensure smooth collaboration with other researchers.

## Creating a new project

1. In order to create a new project in RStudio, click on the R icon with the plus sign in the top left corner of RStudio.

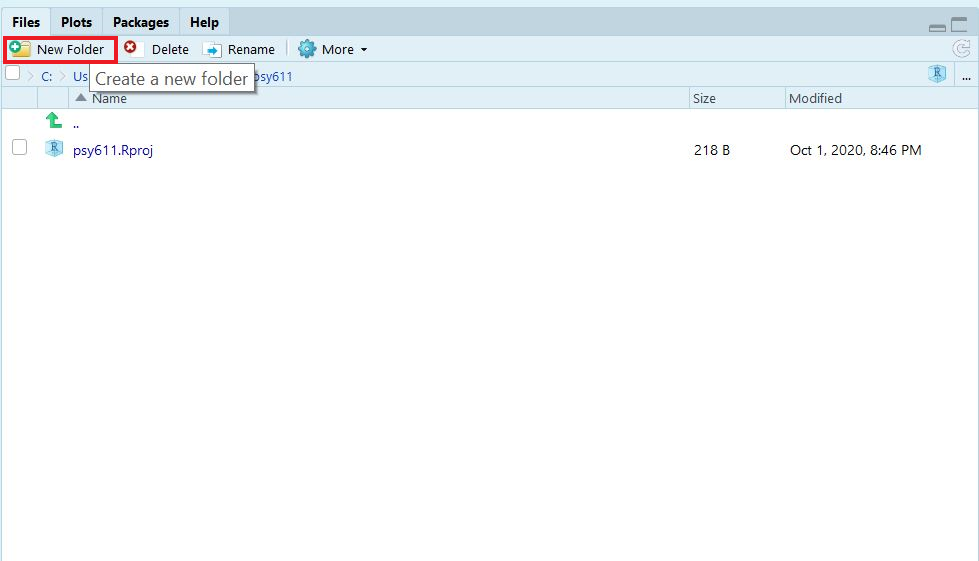


1. Click on New Directory -> New Project. Name your new directory psy611 and store it somewhere on your computer using the Browse button. I would recommend storing it on your desktop.

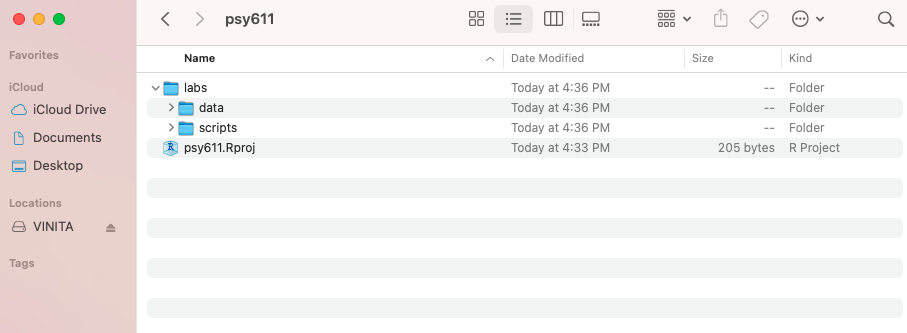


## Adding folders to a project

1. Once you have a new directory, you can add folders to it. I recommend adding a folder for labs and a folder for homeworks since you will need RStudio for both. You can add a folder by clicking on New Folder in the [files/plots/packages/help pane](#rstudio_files).



1. You can nest folders within folders. For example, inside the labs folder, I want to create two more folders: a scripts folder and a data folder.



### Tips for Organizing a Project

* **Use Descriptive Folder Names**: Within your project folder, create subfolders with clear, descriptive names. For example, create folders like data, scripts, figures, and output to store different types of files.
* **Naming Conventions**: Use consistent and descriptive names for your files and folders. This will help you keep track of your work and make it easier to find specific files when your project grows.
* **Backups and Version Control**: Regularly back up your project folder. Consider using version control tools like Git to track changes and collaborate with others.
* **Scripts vs. Output Separation**: Keep your analysis scripts in a separate folder from your data and output files. This ensures that scripts are reusable, and that your data and results stay uncluttered.

### Accessing Your Project

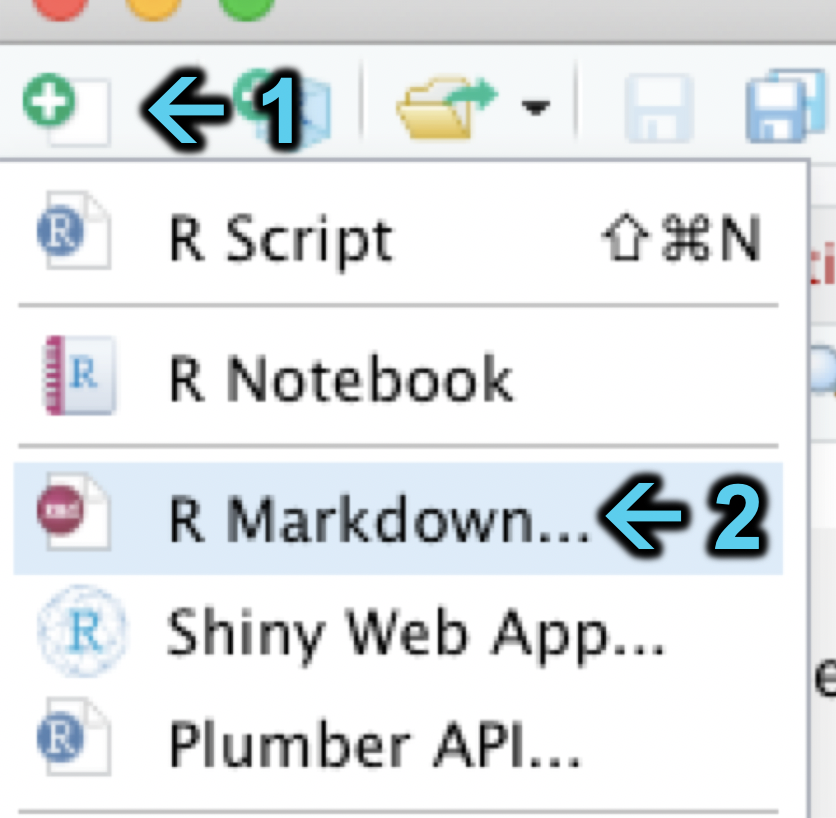
Once a project is set up, you can open it by clicking the .Rproj file located in your project folder. RStudio will open with all your settings and files as they were the last time you worked on the project.

# R Markdown

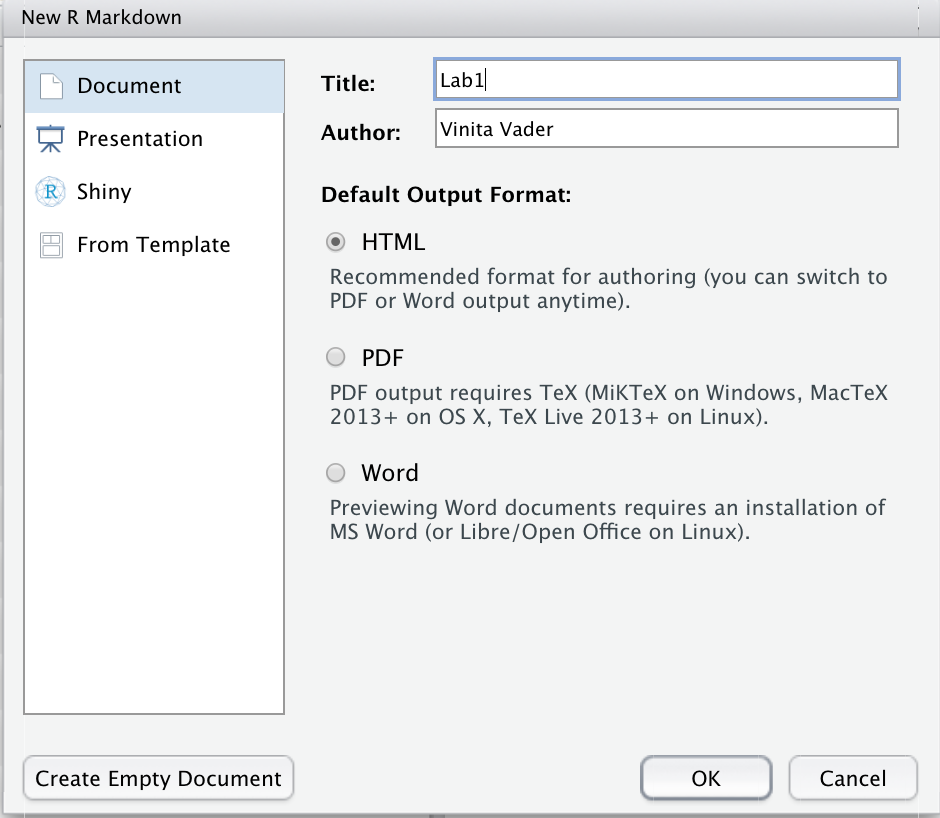
You will mostly be using R Markdown documents in this course. In fact, it is required that your homeworks be created using an R Markdown document. The following section will guide you the process of creating an R Markdown document.

## Creating an R Markdown Document

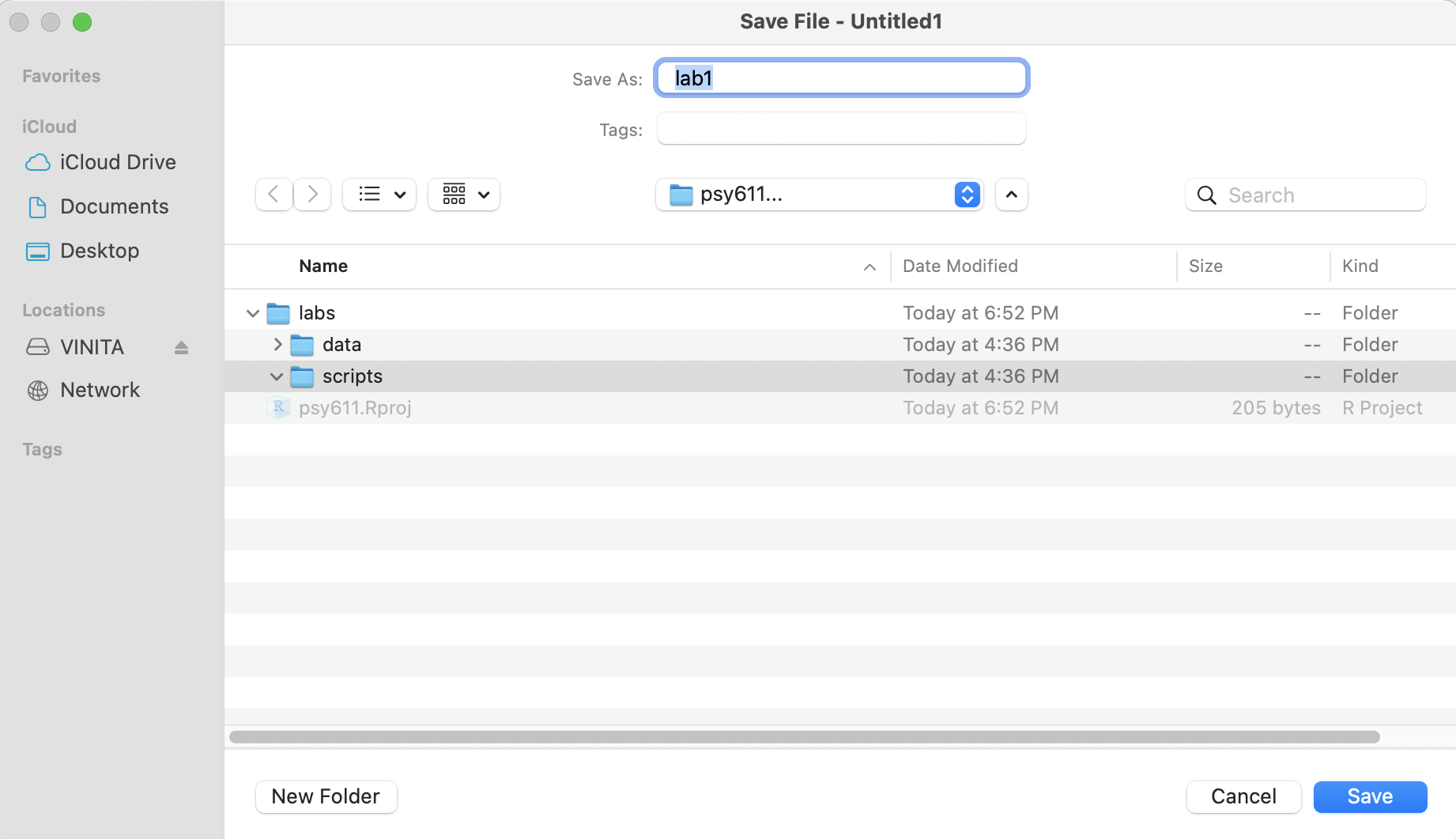
1. Click on the blank piece of paper with the plus sign over it in the upper left-hand corner of RStudio.
2. Click on R Markdown....



1. Enter the title of document and your name. I have chosen to title the document lab1.



1. Save your RMarkdown document by clicking on File -> Save. You want to save it in your labs -> scripts folder in your psy611 project.

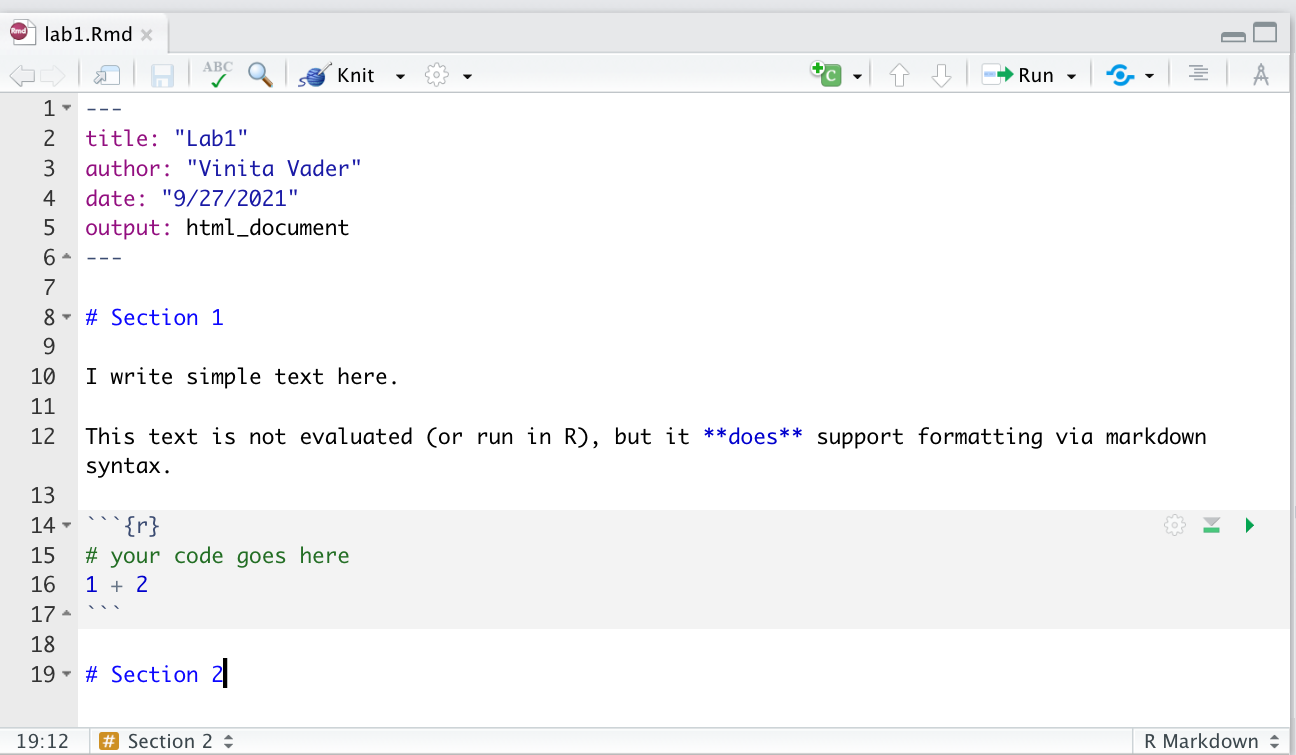


## Using an R Markdown Document

The content of R Markdown documents can be split into two main types. I will call the first type *simple text*. Simple text will not be evaluated by the computer other than to be formatted according to markdown syntax. If you are answering a homework question or interpreting the results of an analysis, you will likely be using simple text.

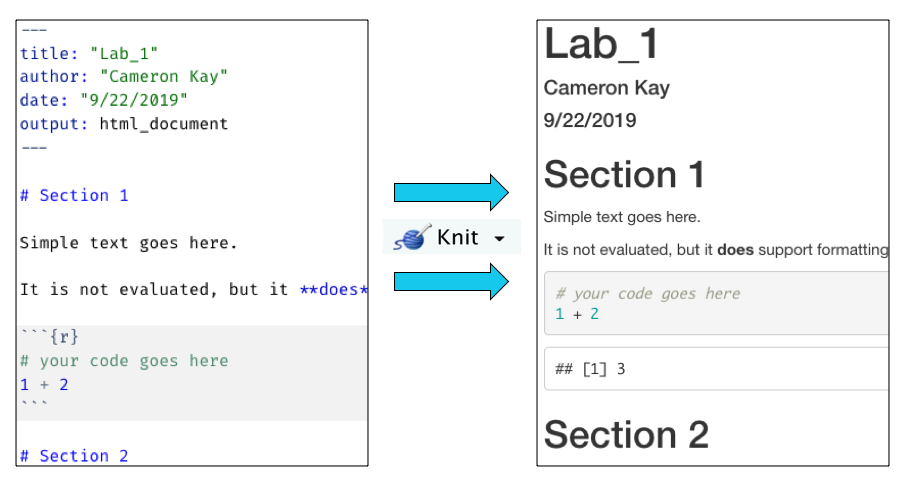
Markdown syntax is used to format the simple text, such as italicizing words by enclosing them in asterisks (e.g., \*this is italicized\* becomes *this is italicized*) or bolding words by enclosing them in double-asterisks (e.g., \*\*this is bold\*\* becomes **this is bold**). For a quick rundown of what you can do with R Markdown formatting, I suggest you check out the Markdown section of the [R Markdown Cheat Sheet](https://rmarkdown.rstudio.com/lesson-15.html).

In addition to simple text, R Markdown documents support blocks (also called chunks) of R code. In contrast to simple text, the R code chunks **are** evaluated by the computer. The chunks are surrounded by ```{r} and ```. In the example image below, the 1 + 2 in the R Code chunk will be evaluated when the document is “knitted” (rendered). For your homeworks, you will want to include your analyses in these chunks.



## Knitting an R Markdown Document

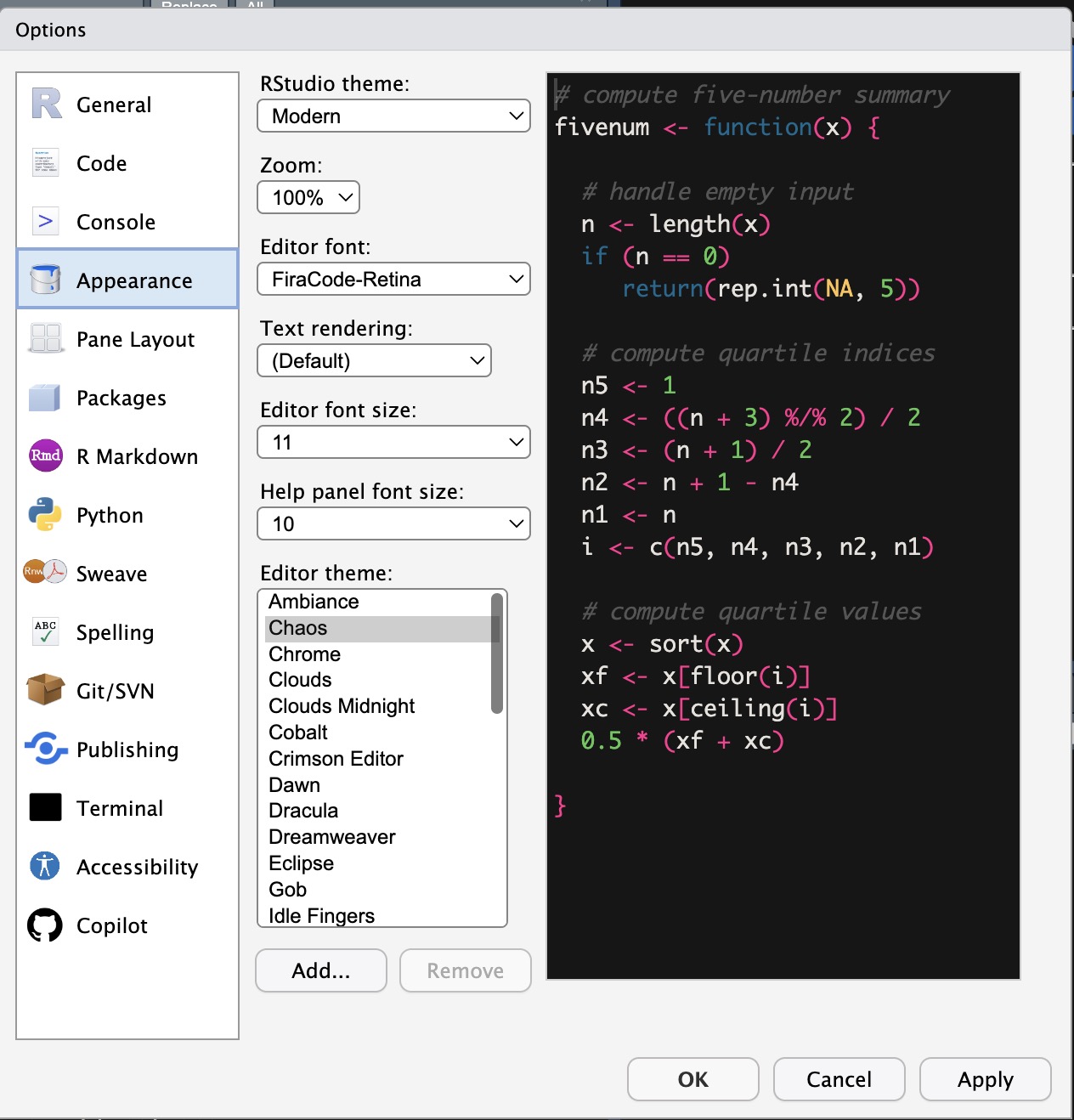
In order to knit an R Markdown document, you can either use the shortcut command + shift + k or click the button at the top of the R Markdown document that says Knit. The computer will take several seconds (or, depending on the length of the R Markdown document, several minutes) to knit the document. Once the computer has finished knitting the document, a new document will appear in the same location that the R Markdown document is saved. In this example, the new document will end with a .html extension.



As shown in the above image, the simple text in the R Markdown document on the left was rendered into a formatted in the knitted document on the right. The equation in the code chunk was also evaluated in the knitted document, returning the value 3.

## Changing RStudio Theme

RStudio has a wide collection of background and font colors. From the top menu bar, click *Tools > Global Options > Appearance*. This will lead you to a screen that looks like this.



As a warning, choose your editor theme wisely, as once you get used to it, changing appearances can totally throw you off.

Note: I use the custom font [firacode-retina](https://github.com/tonsky/FiraCode). This has unique symbols for many of the multi-character symbols we use. It just looks a little cleaner, but is totally personal preference.

# The Basics of Coding in R

## Arithmetic commands

As mentioned above, you can pass commands to the R-engine via the console. R has arithmetic commands for doing basic math operations, including addition (+), subtraction (-), multiplication (\*), division (/), and exponentiation (^).

10 + 5

## [1] 15

10 - 5

## [1] 5

10 \* 5

## [1] 50

10 / 5

## [1] 2

10^5

## [1] 100000

R will automatically follow the PEMDAS order of operations (BEDMAS if you are from Canada or New Zealand or India). Parentheses can be used to tell R what parts of the equation should be evaluated first. As shown below and as expected, (10 + 5) \* 2 is not equivalent to 10 + 5 \* 2.

(10 + 5) \* 2

## [1] 30

10 + 5 \* 2

## [1] 20

## Creating Variables

You can create variables using the assignment operator (<-). Whatever is on the left of the assignment operator is saved to name specified on the right of the assignment operator. I like to imagine that there is a box with a name on it and you are placing a value, inside of the box. For example, if we wanted to place 10 into a variable called my\_number, we would write:

my\_number <- 10

##### Variables with a single numeric value are called scalars

If we want to see what is stored in my\_number, we can simply type my\_number into the console and press enter. We are essentially asking the computer, “What’s in the box with my\_number written on it?”

my\_number

## [1] 10

If we want to overwrite my\_number with a new value, we simply assign a new value to my\_number.

my\_number <- 20

Looking at my\_number again, we can see that it is now 20.

my\_number

## [1] 20

We can treat variables just like we would the underlying values. For example, we can add 5 to my\_number by using +.

my\_number + 5

## [1] 25

Keep in mind, the above operation does not save the result of my\_number + 5 to my\_number. To do that, we would have to assign the result of my\_number + 5 to my\_number.

my\_number <- my\_number + 5  
  
my\_number

## [1] 25

If we want to remove a variable from our environment, we can use rm().

We can create multiple variables that are stored in the environment as long as they have different names.

my\_number\_1 <- 1  
my\_number\_2 <- 2  
my\_number\_3 <- 3  
  
ls() # This function calls everything in your environment

## [1] "my\_number\_1" "my\_number\_2" "my\_number\_3"

If you want to clear your environment, you can use the following function:

rm(list = ls()) # Note that this uses the ls() function described above.  
  
ls() # Check what's currently in the environment

## character(0)

There are generally agreed-upon style guidelines for how to name variables, files, and functions in R. For example, variables are usually named in *snake case*, where words are separated by an underscore (e.g., my\_number). Functions are usually named in *camel case* where words are joined together but capitalized (e.g., writeNewFunction). I highly recommend looking at the [Tidyverse](https://style.tidyverse.org/index.html) guidelines and following them from the beginning of your journey in R - it’ll save you a big headache in a few years.

### Types of Variables

In R, there are four basic types of data: (1) logical values (also called booleans), which can either be TRUE or FALSE, (2) integer values, which can be any whole number (i.e.., a number without digits after the decimal place), (3) double values, which can be any number with digits before and after the decimal place, and (4) character values (also called strings), which are pieces of text enclosed in quotation marks (").

| Type | Examples |
| --- | --- |
| Logical/Boolean | TRUE, FALSE |
| Integer | 10L, -10L |
| Double | 10.50, -10.50 |
| Character | "Hello", "World" |

###### The “L” after the number in integers comes from the C programming language, where it stands for long.

### Vectors

#### Atomic Vectors

A collection of values is called a vector. If they are all of the same type, we call them atomic vectors. In R, we use the c() command to concatenate (or combine) values into an atomic vector.

c(10, 20, 20, 40, 50, 60)

## [1] 10 20 20 40 50 60

Just as we did with the scalar values above, we can assign a vector to a variable.

my\_vector <- c(10, 20, 20, 40, 50, 60)

To print out the entire vector, we simply type my\_vector into the console.

my\_vector

## [1] 10 20 20 40 50 60

In order to select just one value from the vector, we use square brackets ([]). For example, if we wanted the third value from my\_vector we would type my\_vector[3][1](#foot_vector).

my\_vector[3]

## [1] 20

If we want to replace a specific value in a vector, we use the assignment operator (<-) in conjunction with the square brackets ([]).

my\_vector[3] <- 30  
my\_vector

## [1] 10 20 30 40 50 60

As with single-value objects we can perform arithmetic operations on vectors, but the behavior is not identical. If the vectors are the same length, each value from one vector will be paired with a corresponding value from the other vector. See below for an example of this in action.

my\_vector\_2 <- c(1, 2, 3, 4, 5, 6)  
  
my\_vector + my\_vector\_2

## [1] 11 22 33 44 55 66

If the vectors of different lengths, the shorter vector will be recycled (i.e., repeated) to be the same length as the longer vector.

my\_vector\_3 <- c(1000, 2000)  
  
my\_vector + my\_vector\_3

## [1] 1010 2020 1030 2040 1050 2060

This also works when the longer vector is not a multiple of the shorter vector, but you will get the warning: longer object length is not a multiple of shorter object length.

my\_vector\_4 <- c(1000, 2000, 3000, 4000)  
  
my\_vector + my\_vector\_4

## Warning in my\_vector + my\_vector\_4: longer object length is not a multiple of  
## shorter object length

## [1] 1010 2020 3030 4040 1050 2060

###### 1. Unlike most other coding languages (e.g., python), indices in R start at 1 instead of 0. For instance, if you want to select the first element of a vector, you would write my\_vector[1] instead of my\_vector[0]. A second difference to keep in mind is that the - is used in R to remove whichever value is in the spot indicated by the index value. Using vector[-2] on the vector c(10, 20, 30, 40, 50, 60) would return c(10, 30, 40, 50, 60) in R. In python, it would return 50.

#### Lists

A vector that can accomodate more than one type of value (e.g., a double AND a character) is called a list. To create a list, we use list() instead of c(). If we wanted to create a vector with the values 5L, 10, "fifteen", and FALSE we would use list(5L, 10, "fifteen", FALSE).

list(5L, 10, "fifteen", FALSE)

## [[1]]  
## [1] 5  
##   
## [[2]]  
## [1] 10  
##   
## [[3]]  
## [1] "fifteen"  
##   
## [[4]]  
## [1] FALSE

Although lists are an incredibly powerful type of data structure, dealing with them can be quite frustrating (especially for beginning coders). Since you are unlikely to need to know the inner workings of lists for anything we will be doing in this course, I have chosen not to include much about them here. However, as you become a more advanced user, learning to leverage lists will allow you to write code that is far more efficient.

### Data Frames

In R you will mostly be working with data frames. A data frame is technically a list of atomic vectors. For our purposes, we can think of a data frame as a spread sheet with columns of variables and rows of observations.

Let’s look at a data frame that is automatically loaded when you open R, mtcars. Type mtcars to print out the data frame.

## 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

The data frame mtcars has a row for 32 cars featured in the *1974 Motor Trend* magazine. There is a column for the car’s miles per gallon (mpg), number of cylinders (cyl), engine displacement (disp), horse power (hp), rear axle ratio (drat), weight in thousands of pounds (wt), quarter-mile time (qsec), engine shape (vs), transmission type (am), number of forward gears (gear), and number of carburetors (carb).

With data frames, you can extract a value by including [row, col] immediately after the object. For example, if we wanted to extract the number of gears in the Datsun 710 we could use mtcars[3, 10] to extract the value stored in the third row, tenth column.

mtcars[3, 10]

## [1] 4

Since the rows and columns have names, we can also be explicit and use the name of the row ("Datsun 710") and the name of the column ("gear") instead of the row and column indices.

mtcars["Datsun 710", "gear"]

## [1] 4

We can also extract an entire column by dropping the index value for the row. Since you don’t specify a given row, the computer assumes you want all of the values in the column. For example, to extract all values stored in the gear column, we could use [, 10] or [, "gear"].

mtcars[, 10]

## [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4

mtcars[, "gear"]

## [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4

To extract an entire row, we drop the column index. To extract all of the values associated with the Datsun 710, we would drop the column index (e.g., [3, ] or ["Datsun 710", ])

mtcars[3, ]

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

mtcars["Datsun 710", ]

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

You can also extract columns using $ followed by the column name without quotes.

mtcars$gear

## [1] 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3 3 3 4 4 4 3 3 3 3 3 4 5 5 5 5 5 4

If we want to extract multiple columns (or multiple rows) we use vectors. For example, if we wanted the number of gears and carburetors in the Datsun 710 and the Duster 360 we would use [c("Datsun 710", "Duster 360"), c("gear", "carb")] or [c(3, 7), c(10:11)].

mtcars[c("Datsun 710", "Duster 360"), c("gear", "carb")]

## gear carb  
## Datsun 710 4 1  
## Duster 360 3 4

print(mtcars[c(3, 7), c(10:11)])

## gear carb  
## Datsun 710 4 1  
## Duster 360 3 4

## Functions

Up to this point, we have been more-or-less directly telling R what we want it to do. This is great if we want to understand the processes that underlie R, but it can be incredibly time-consuming. Thankfully, we have functions. Functions are essentially pre-packaged snippets of code that take one or more pieces of input (called arguments) and return one or more pieces of output (called values). For example, length() is a function that takes a vector as its sole argument and returns the length of the vector as its sole value.

length(c(10, 20, 30, 40, 50, 60))

## [1] 6

The function unique() also takes a vector as its primary argument, but—instead of returning the length of the vector as its value—it returns only the unique values of that vector.

unique(c("cond\_a", "cond\_a", "cond\_b", "cond\_a", "cond\_b"))

## [1] "cond\_a" "cond\_b"

The mean() function and sd() function are two functions that you will end up using a lot. The former (mean()) takes a numeric vector and returns the average of the vector.

mean(c(10, 20, 30, 40, 50, 60))

## [1] 35

The latter (sd()) also takes a numeric vector, but it returns the standard deviation of the vector instead.

sd(c(10, 20, 30, 40, 50, 60))

## [1] 18.70829

Although it is more conceptual, it is also useful to mention the typeof() function here. The function typeof() takes any object and tells you what type of variable it is.

typeof(10L)

## [1] "integer"

typeof(10)

## [1] "double"

typeof("hello")

## [1] "character"

typeof(TRUE)

## [1] "logical"

Using the suite of as.\*() functions (e.g., as.numeric(), as.character(), as.logical(), as.integer()), we can likewise coerce objects to other types.

as.numeric("10")

## [1] 10

as.character(10)

## [1] "10"

as.logical(1)

## [1] TRUE

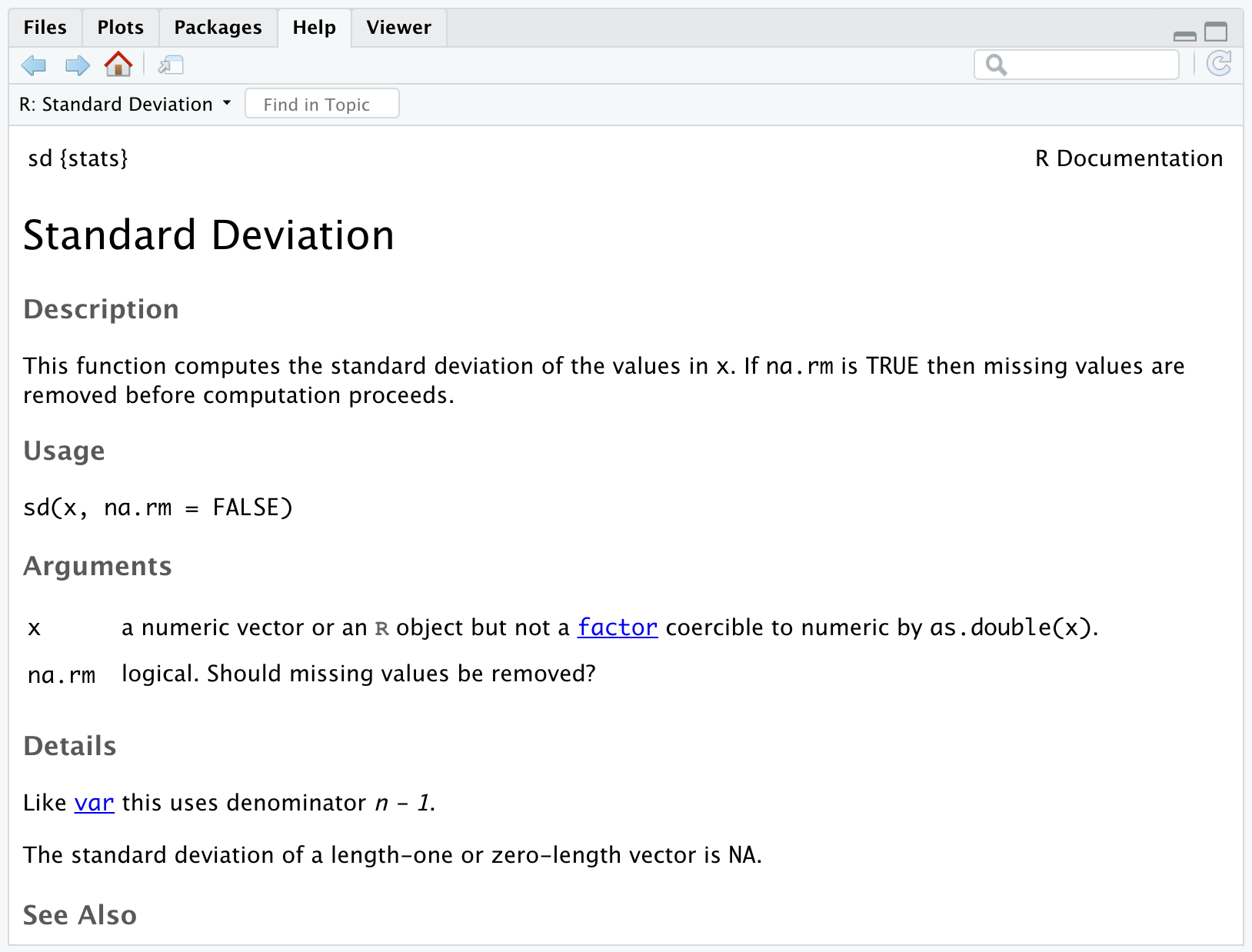
as.integer(10.30)

## [1] 10

## Help Documentation

Sometimes when working in R you will want to know more about a function. For example, you might want to know what arguments the function sd() takes. You can use ? at the beginning of any function call to display the help documentation for that function.





From the help documentation we can see that sd() takes two arguments: (1) An R object and (2) a logical value indicating whether NAs (unknown values) should be removed before the standard deviation is calculated.

Typically R will infer, based on the order of the arguments, what values correspond to which arguments. For example, since sd() expects that the argument x will be provided first and the argument na.rm will be provided second, the following works:

sd(c(10, 20, 30, 40, 50, 60), FALSE)

## [1] 18.70829

However, we can also explicitly tell R what values are associated with which arguments.

sd(c(10, 20, 30, 40, 50, 60), na.rm = FALSE)

## [1] 18.70829

The help documentation for a function often also includes an example of how to use the function and details on what the expected output will be.

## Googling your error message

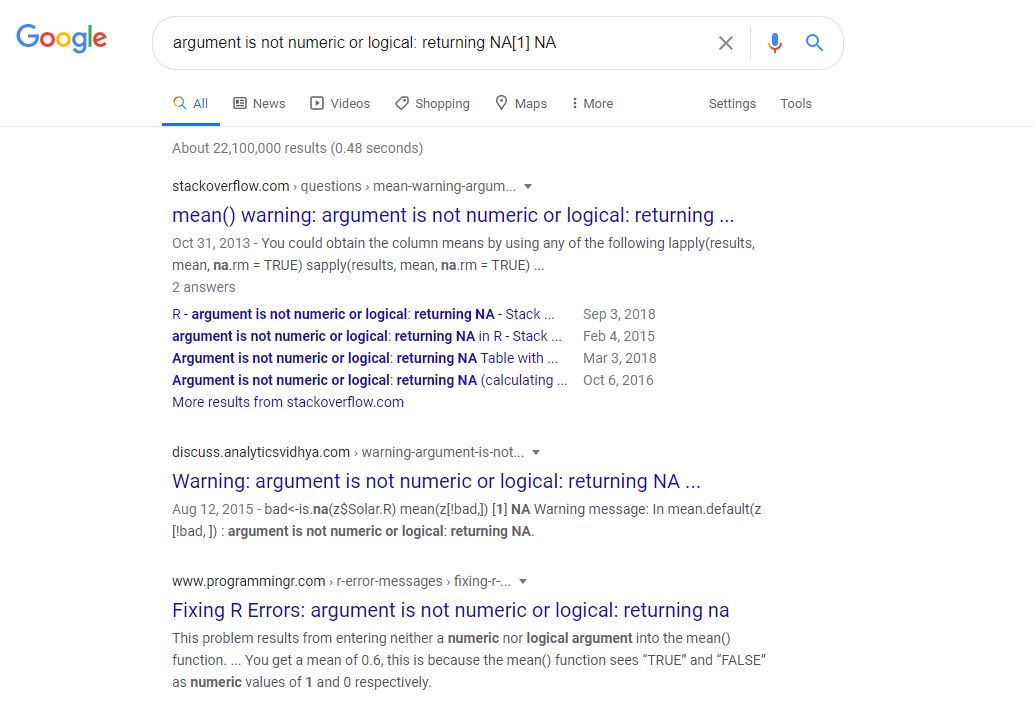
You will come across many messages in your time using RStudio. Some messages are error messages and some are warning messages. If a message says warning message then R was able to run the code but not as it was intended. An error message means that R was not able to run the code at all. Here is an example of code that would produce a warning message.

mean(c(4,5,"6",7,5))

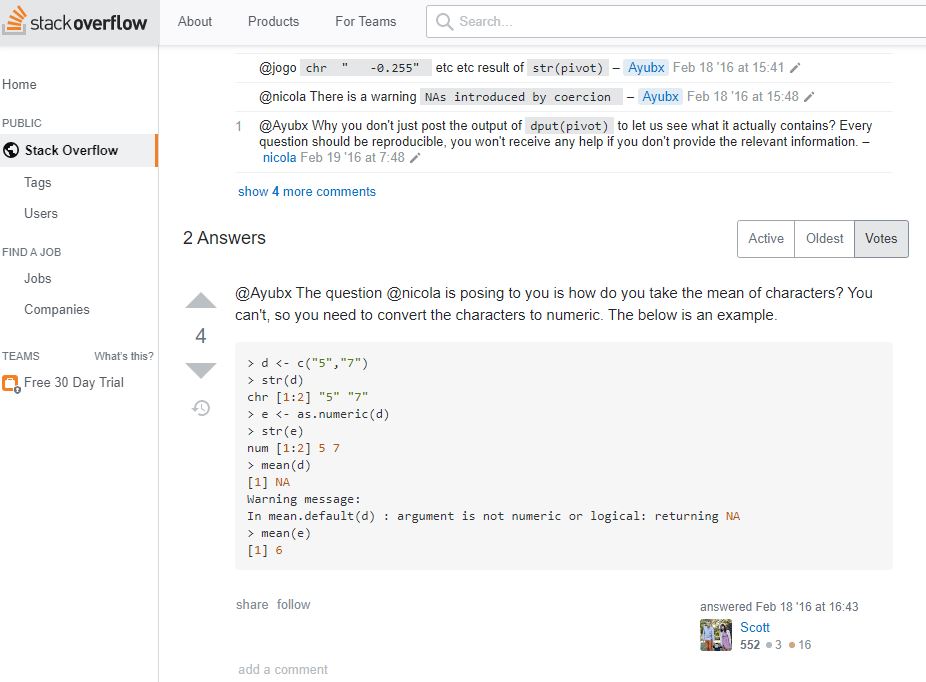
## Warning in mean.default(c(4, 5, "6", 7, 5)): argument is not numeric or  
## logical: returning NA

## [1] NA

When you get a warning or error message, and you aren’t sure what it means, you should first try googling the message. Oftentimes, others have encountered your problem and have asked for help deciphering the message.



Scott from Stack overflow suggests converting the character “6” into a numeric variable. Let’s try that.



mean(c(4,5,as.numeric("6"),7,5))

## [1] 5.4

## Comments

Comments are pieces of code text that are not interpreted by the computer. In R we use the octothorpe/pound sign/hashtag (#) at the beginning of a line to denote a comment. The first and third line of code below are not evaluated, whereas the second and fourth line are.

## [1] 4

## [1] 8

Comments are mostly used to remind yourself (or other people) *what* a piece of code does and *why* the code is written the way that it is. Below is a piece of code that checks if a string is a valid phone number. We can see that the comments explain, not only what each piece of code is doing, but also why the second piece of code was written the way that it was.

# assign the phone number  
phone\_number <- "(541)-346-4921"  
  
# validate the phone number; I did not account for country codes because all of the numbers are either from the US or Canada  
grepl("^\\(?(\\d{3})\\)?(-| )\*(\\d{3})(-| )\*(\\d{4})$", phone\_number)

## [1] TRUE

# Minihacks

Now that we have covered the lab material, we will move on to the Minihacks. If you have any questions, I would be happy to answer them!

## Minihack 1: R Markdown

1. Create an R Markdown document called lab1\_minihacks. Save it in your Lab -> scripts folder.
2. Try rendering your R Markdown document by clicking knit. If it doesn’t render correctly, try to figure out why it didn’t.

## Minihack 2: Arithmetic Commands

1. Use R to calculate and assign the result to a variable called x.
2. Assign the numbers 10, 20, and 30 to a vector called y.
3. Before running any code, determine what you think adding x to y would result in. Then, using R, add x to y.

## Minihack 3: Functions

1. Assign the string "I AM NOT YELLING" to a variable called exclamation.
2. Use the function tolower() to convert every letter of exclamation to lower case. Assign the result to exclamation.
3. Use the capitalize() function from the Hmisc package to capitalize the first letter of exclamation.

## Minihack 4: Help Documentation

1. I wanted to create a vector of 5 values between 10 and 50 using seq(), but the code I wrote is creating a vector of 9 values between 10 and 50. I believe it has something to do with the arguments I used, but I can’t remember how to access the help documention to check. Without changing the values (i.e., 10, 50, and 5), can you fix my code?

seq(from = 10, to = 50, by = 5)

## [1] 10 15 20 25 30 35 40 45 50

## Minihack 5: Data Frames

1. Download the [Marvel character dataset](https://docs.google.com/spreadsheets/d/1n59gFMfvLXN1rUwRjFV5qf5VNp3im_bj6AgHuv7mXTs/export?format=xlsx) to your computer. Put the data file in your labs -> data folder.
2. Import the data into R and assign it to a variable called marvel\_data.
3. Ah! The value for the number of appearances of Spider-Man seems to be an error! It should be 4043 not 40430! Use square brackets ([]) to replace the erroneous value with the correct value (*hint:* The value is stored in the first row of the eighth column).
4. Using mean() and dollar sign notation (data$column), calculate the average number of appearances for all of the Marvel characters. Assign the result to a variable called mean\_appearances.
5. Install and load the package ggplot2. If you have installed the tidyverse package as part of the R bootcamp, ggplot2 should already be installed.
6. If you successfully completed the proceeding steps, you should be able to run the following code without producing an error. If you get an error, try to figure out why you are receiving the error.

ggplot(marvel\_data, aes(x = reorder(align, -appearances),   
 y = appearances,  
 fill = align)) +  
 geom\_bar(stat = "summary", fun.y = "mean") +  
 geom\_point(shape = 21,  
 alpha = .7,  
 position = position\_jitter(w = 0.4, h = 0)) +  
 geom\_hline(yintercept = mean\_appearances,  
 linetype = "twodash",  
 lwd = 1,  
 colour = "firebrick") +  
 annotate(geom = "text",   
 x = 3,   
 y = 800,   
 size = 5,  
 label = paste("Mean = ", round(mean\_appearances, 2)),  
 colour = "firebrick") +  
 scale\_fill\_viridis\_d() +  
 theme\_bw(base\_size = 15) +  
 theme(legend.position = "none") +   
 labs(title = "Alignment and Appearances",  
 subtitle = "Marvel character appearances by alignment",  
 x = "Alignment",  
 y = "Appearances")