Lab 1: Introduction to R and RStudio

Contents

You can download the .Rmd file here. You can download the .doc file here.

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:

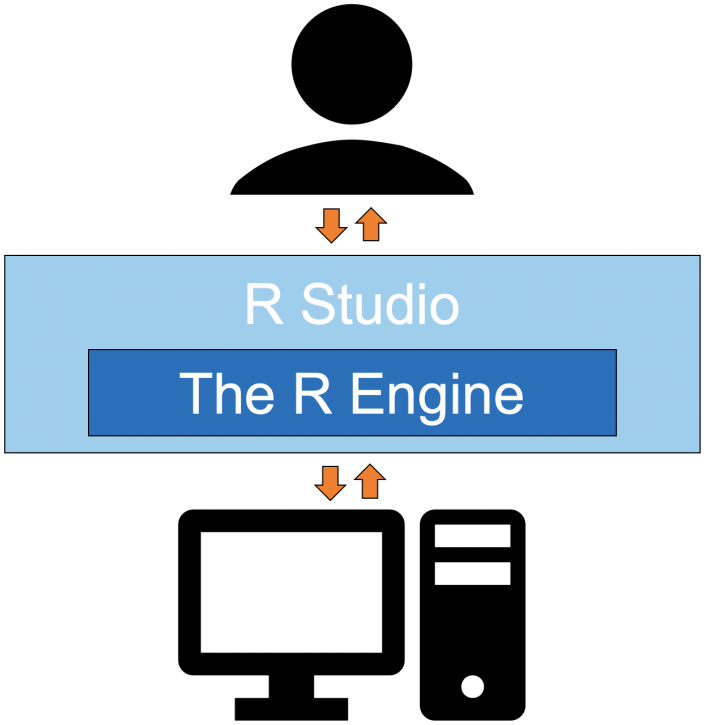
1. How to download and install R and RStudio
2. The panes of RStudio
3. How to create and use R Markdown Documents
4. Arithmetic commands
5. (Some of) the different types of variables in R
6. What functions are and how to use them
7. How to install and load packages and...
8. How to import data into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

After we have covered the content of the lab, we will move on to 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 are below.

A second piece of software that is not required to use R but is nonetheless useful is RStudio. RStudio 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 are also below.

Downloading the R Engine

1. Navigate to the webpage for the Comprehensive R Archive Network (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.
3. Click the link for the latest release. As of writing this (September, 2021), the newest version is `R 4.1.1. "Kick Things"` (all version nicknames are references to the Peanuts comic strip). I would click `R-4.1.1.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.

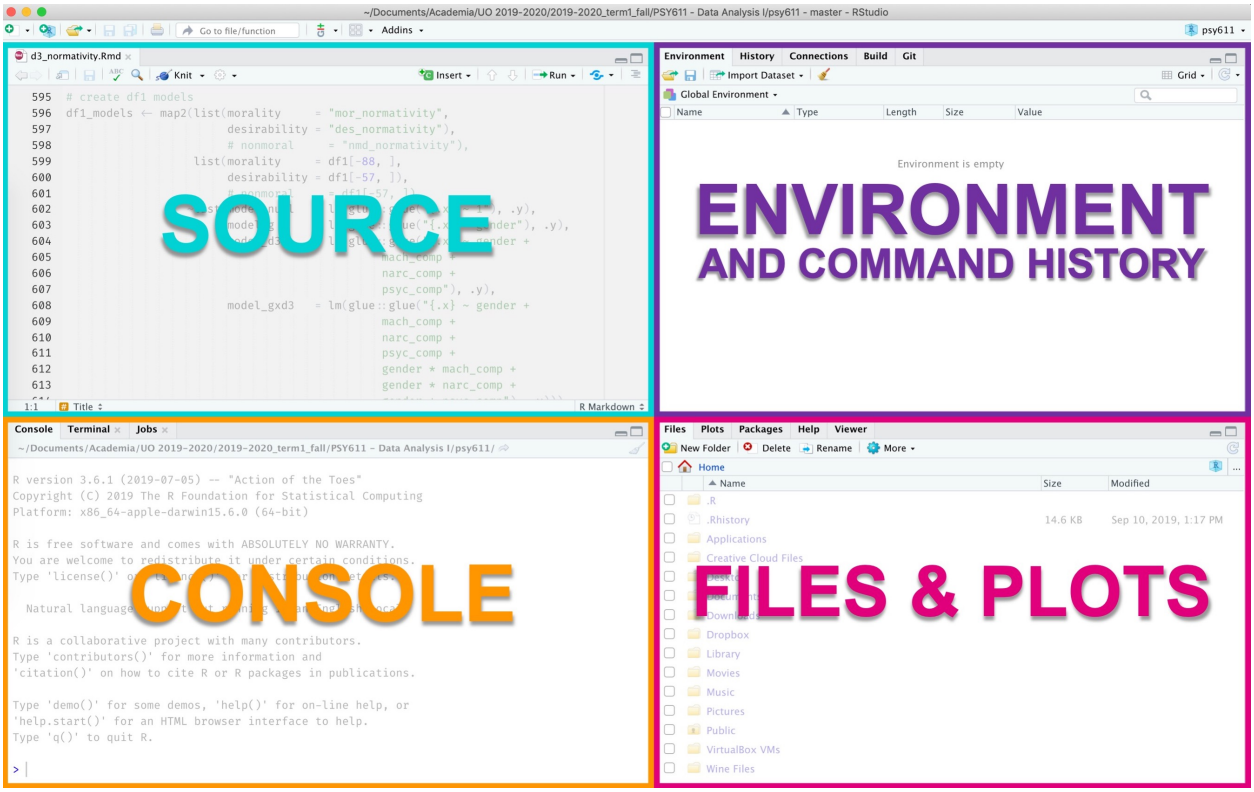
Downloading RStudio

1. Navigate to the webpage for the free version of RStudio. For our purposes (and for most people's purposes) the free version of RStudio 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-1.4.1717.dmg`. If you are using Windows 10, you would click `RStudio-1.4.1717.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 after it is already installed, all you have to do is navigate to `Help > Check for Updates` in the menubar.

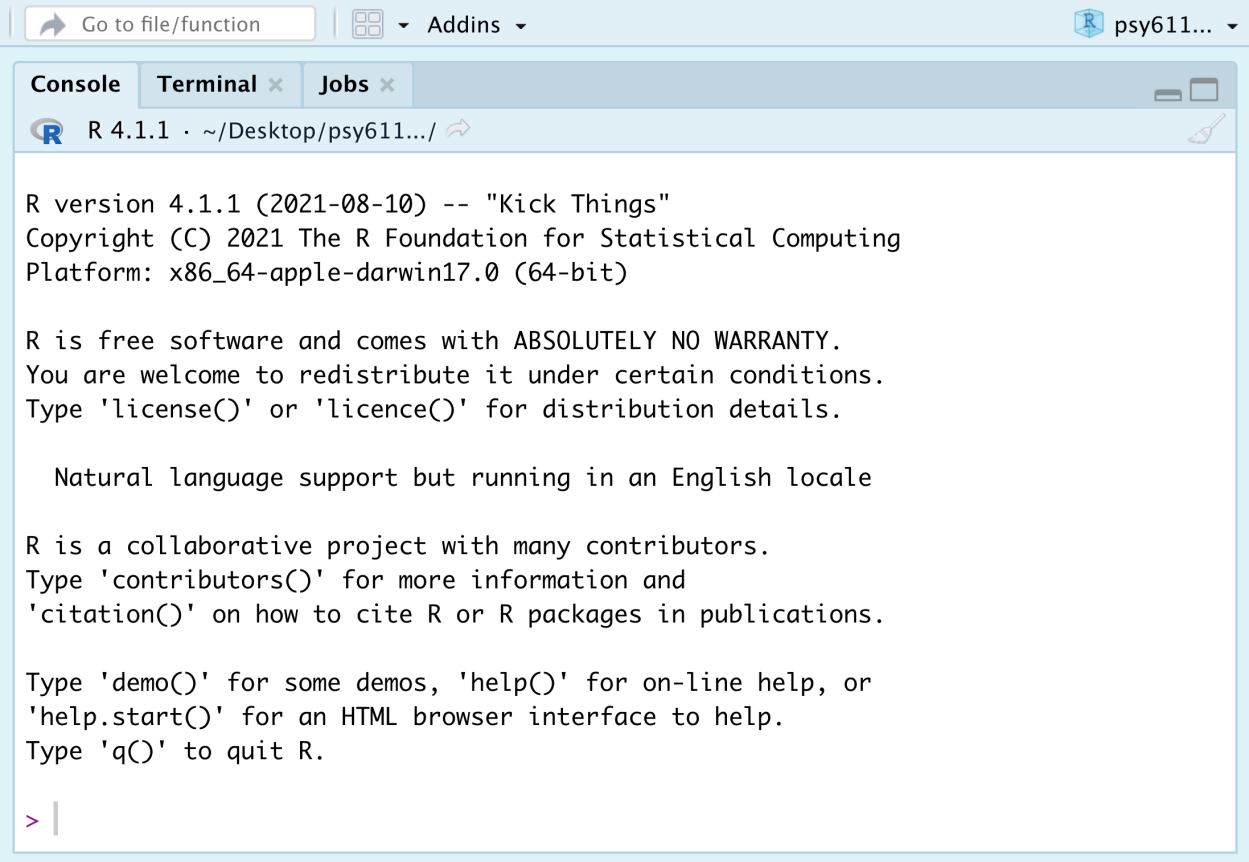
Features of RStudio

As shown in the image below, an RStudio session is split into four sections called panes: the console, the source pane, the environment/history pane, and the succinctly named files/plots/packages/help pane.



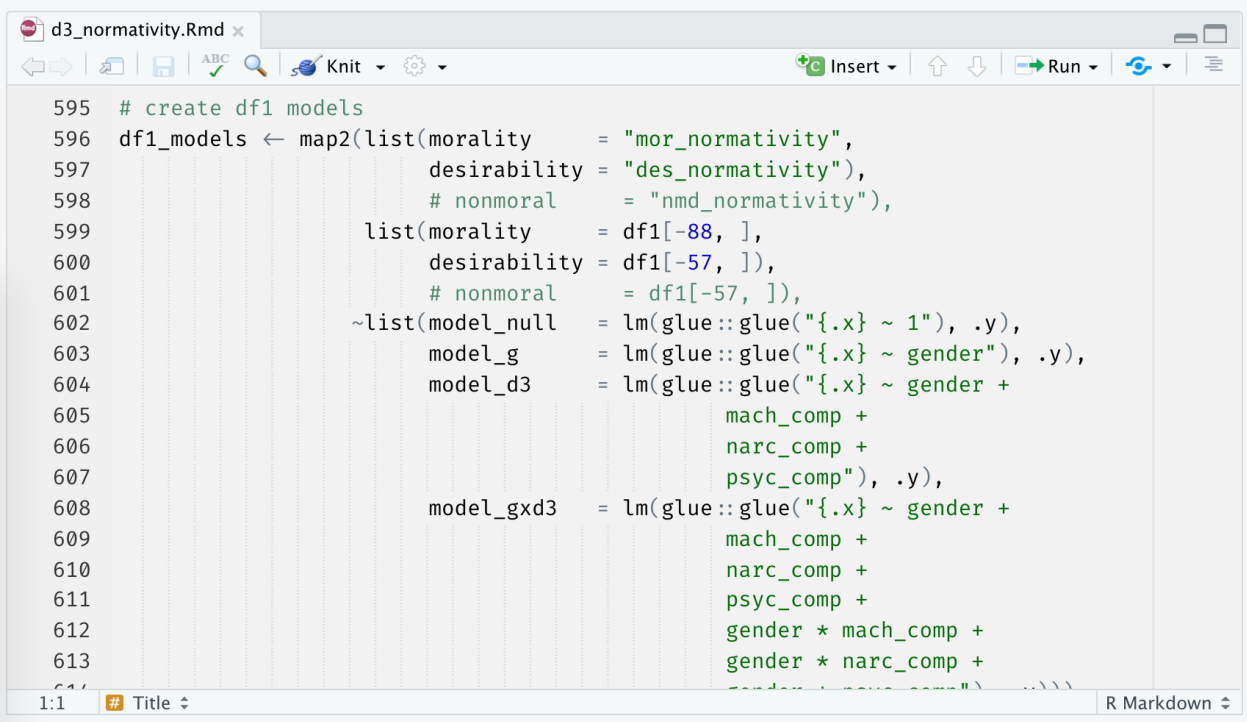
The Console

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. You can pass commands to the R Engine by typing them in after the >.



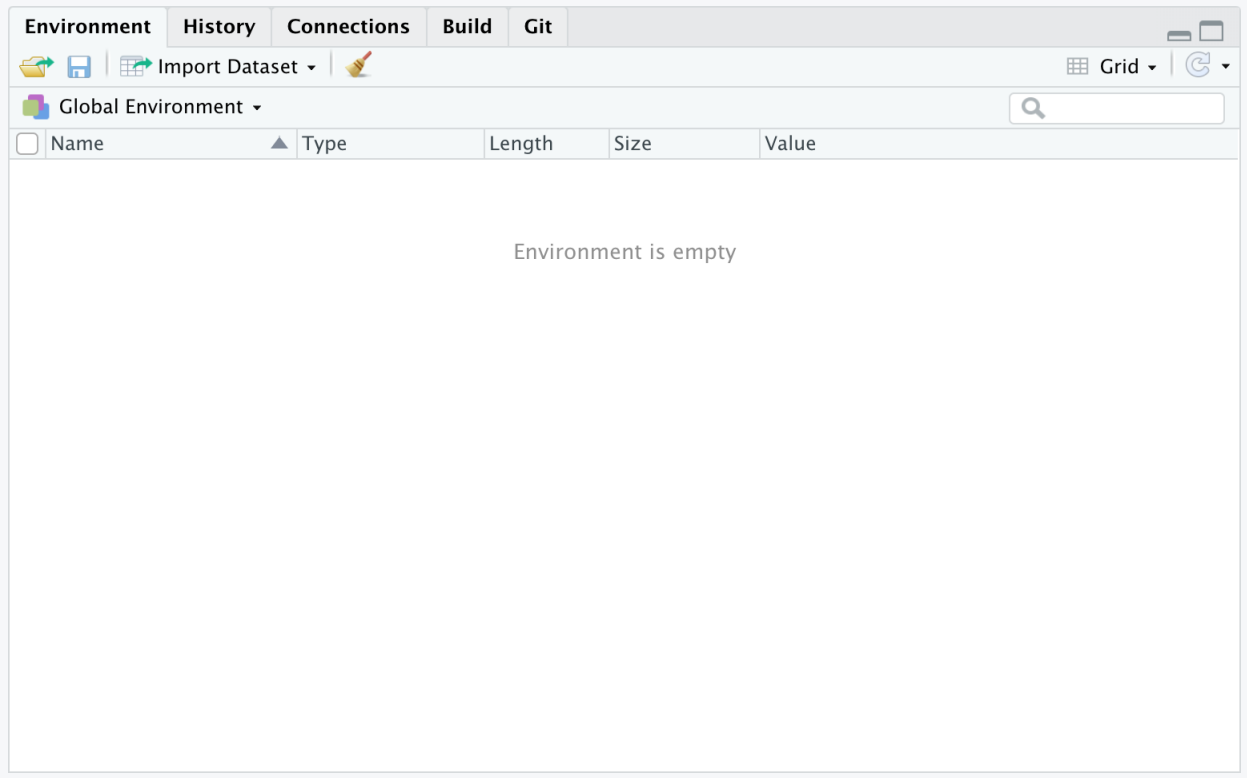
Source

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.



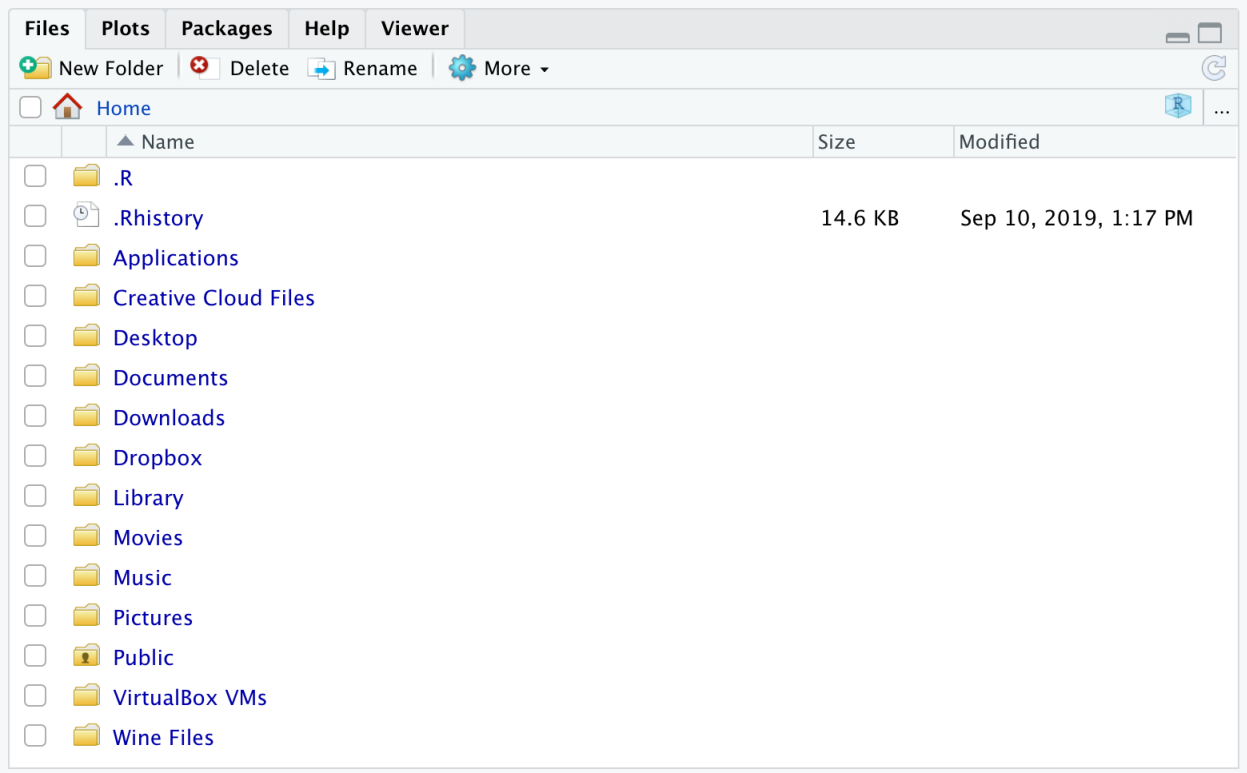
Environment/History

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.

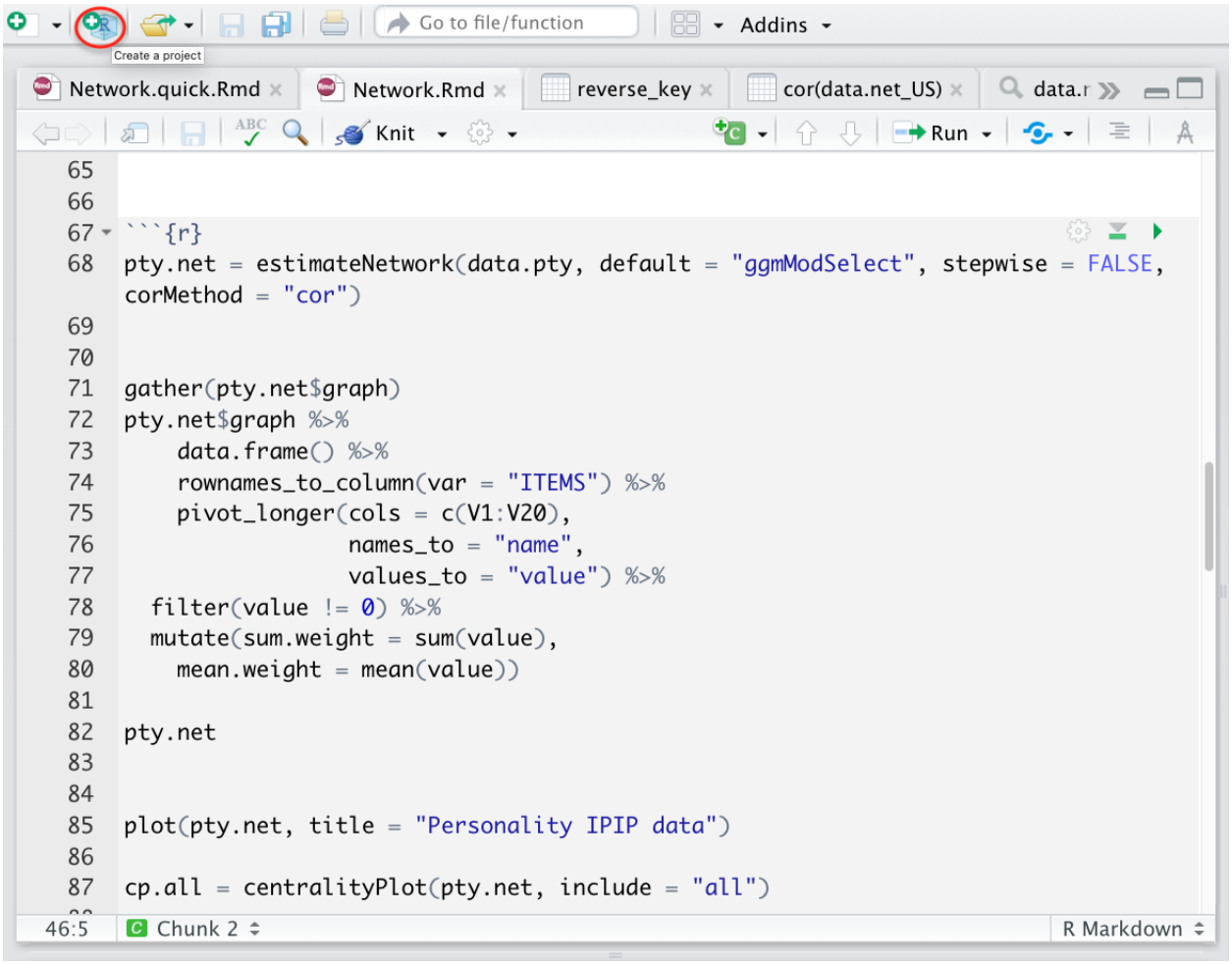


Projects

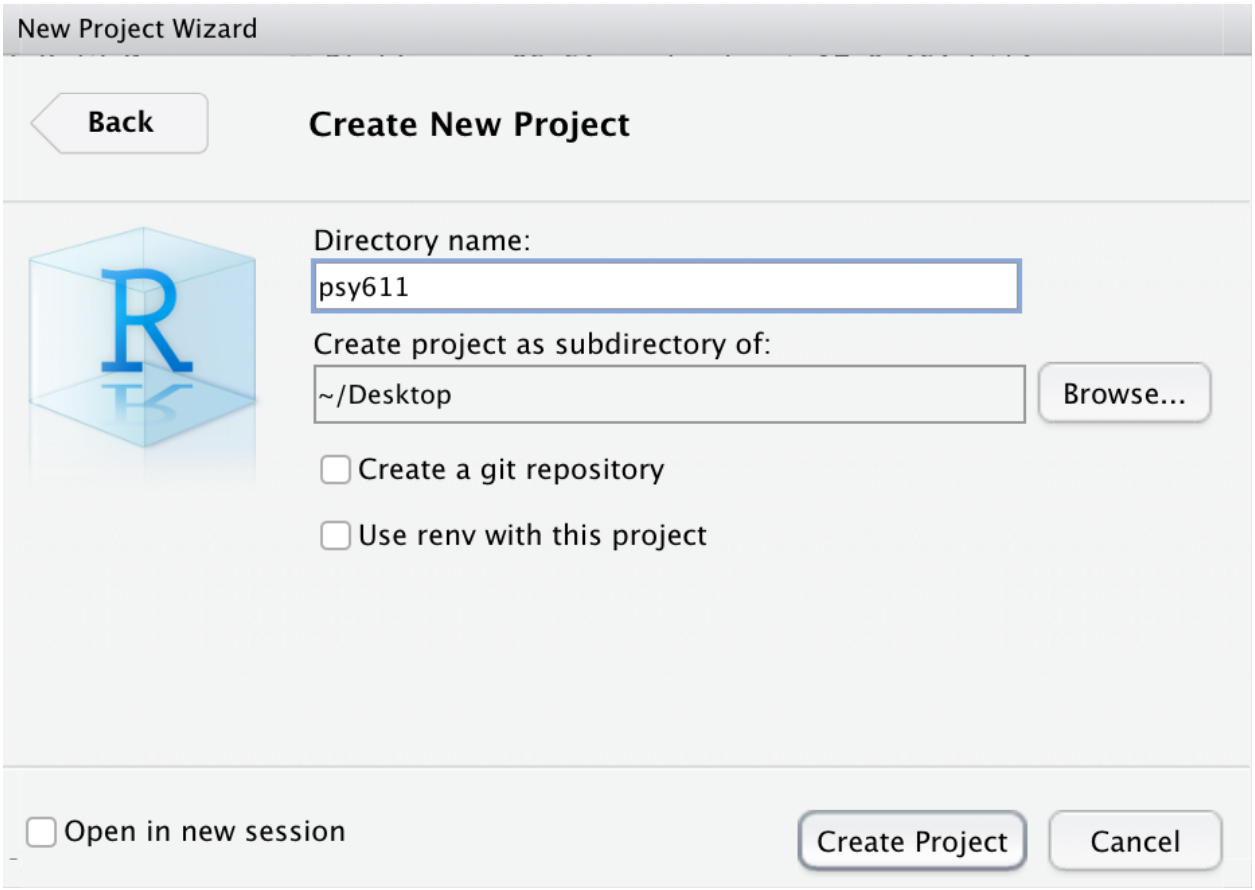
Whenever you start a new research project, you should create a new *R Project.* The R project is a working directory where your `.RProj` file, scripts, data, images, etc. will live. Creating a folder that contains all of the files for your new research project will keep you organized and make it easy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

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.

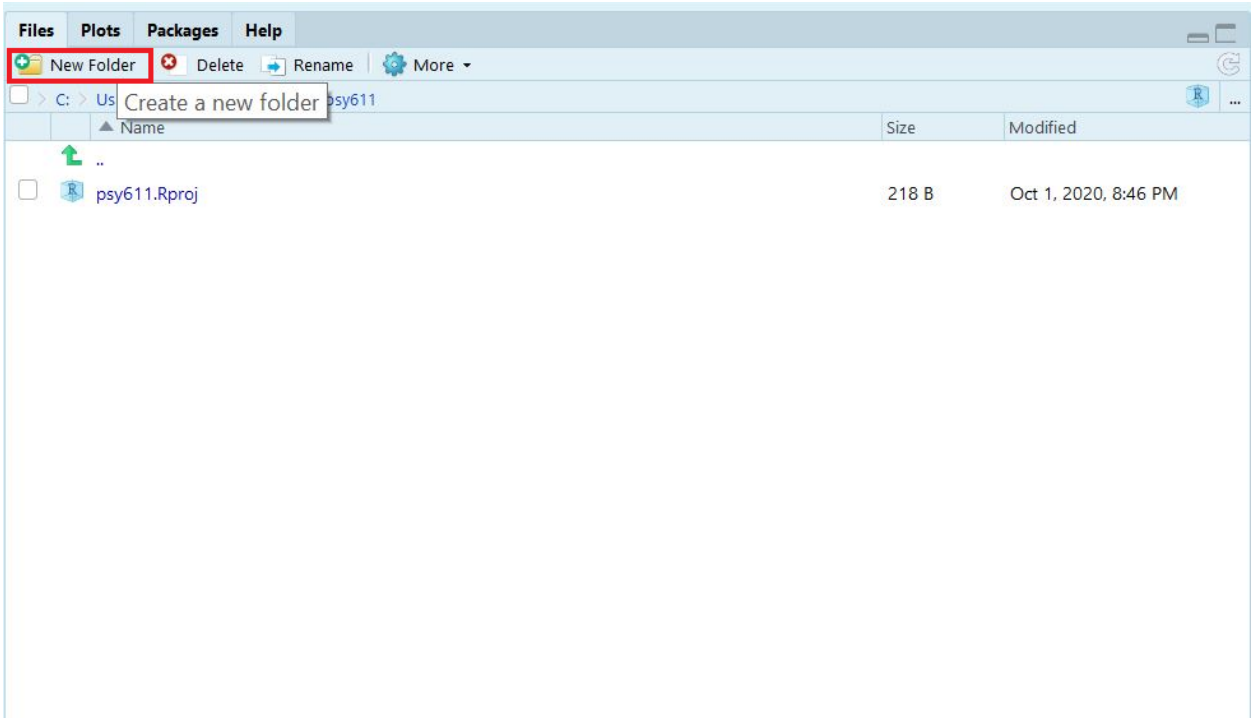


2. Click on `New Directory -> New Project`. Name your new directory `psy611` and store it somewhere on your computer using the `Browse` button. I would recommend storing it on your desktop.

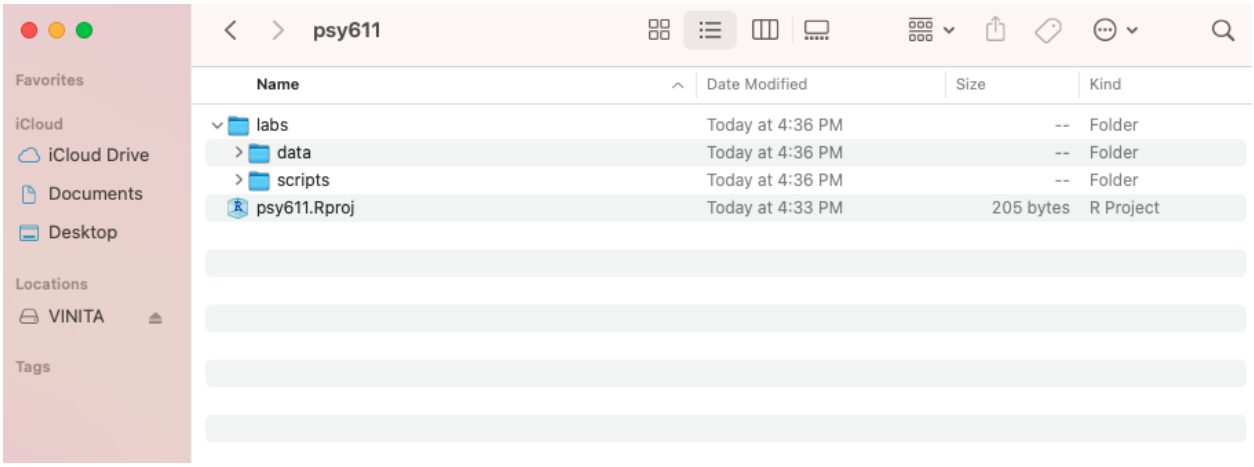


Adding folders to a project

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.



2. 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.

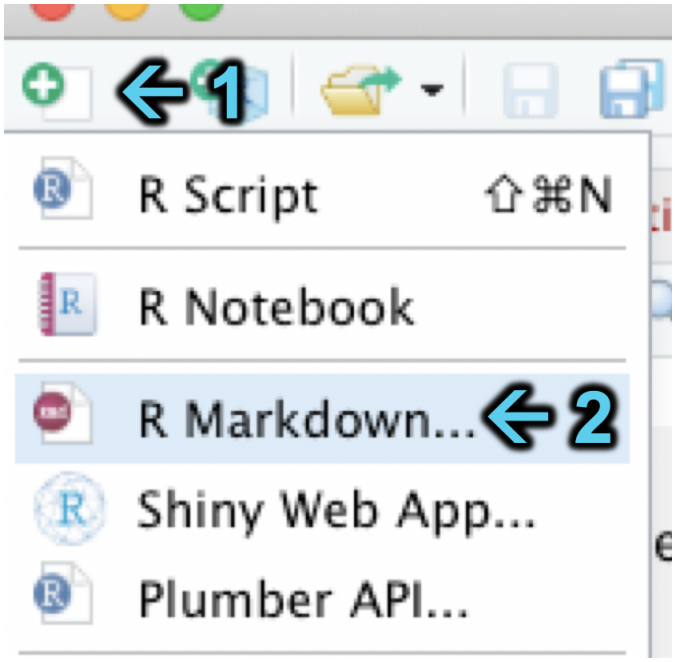


R Markdown

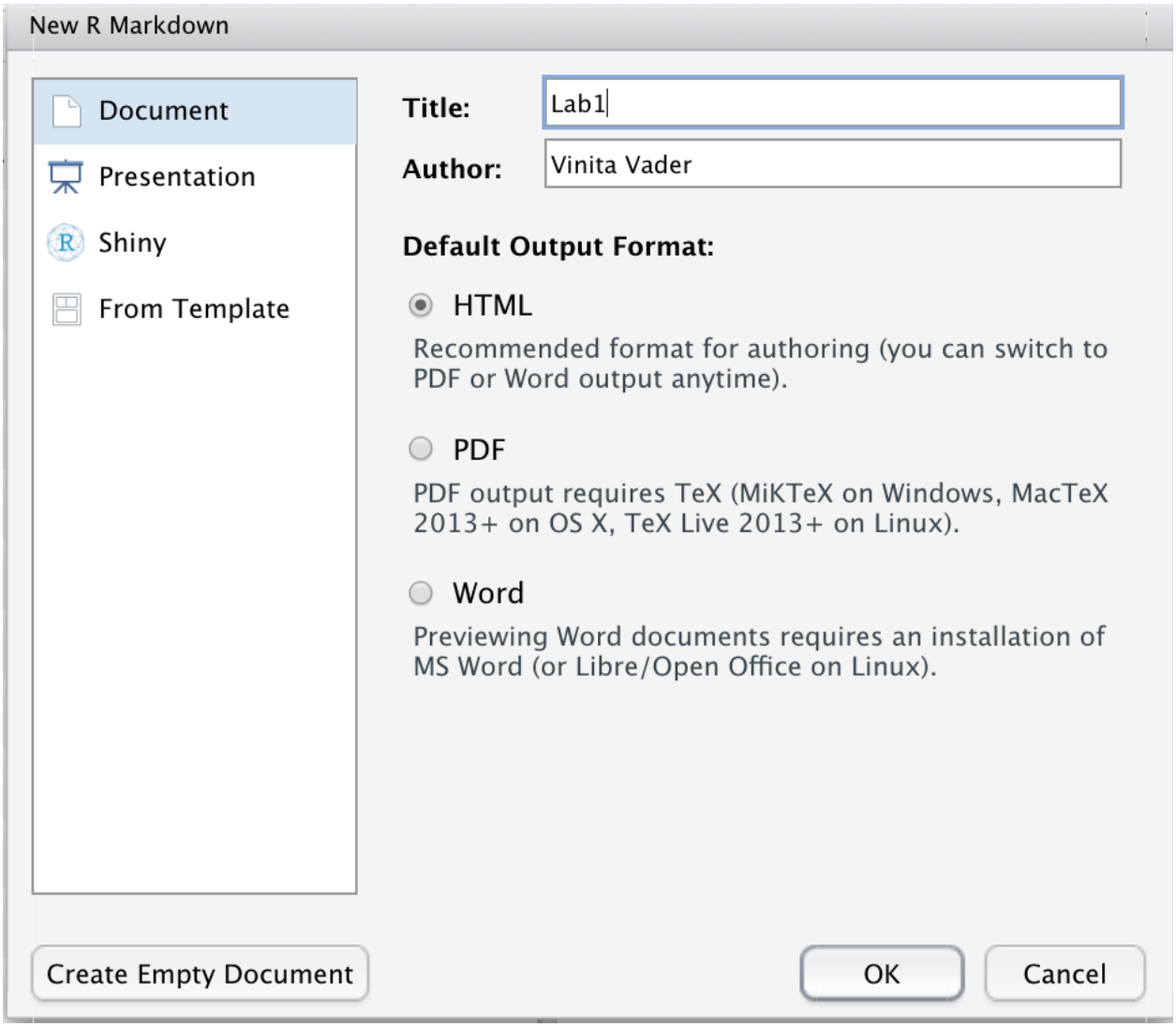
You will mostly be using `R Markdown` documents in this course. In fact, it is required that your homeworks be created using an `R Markdown` document. The following section will guide you the process of creating an `R Markdown` document.

Creating an R Markdown Document

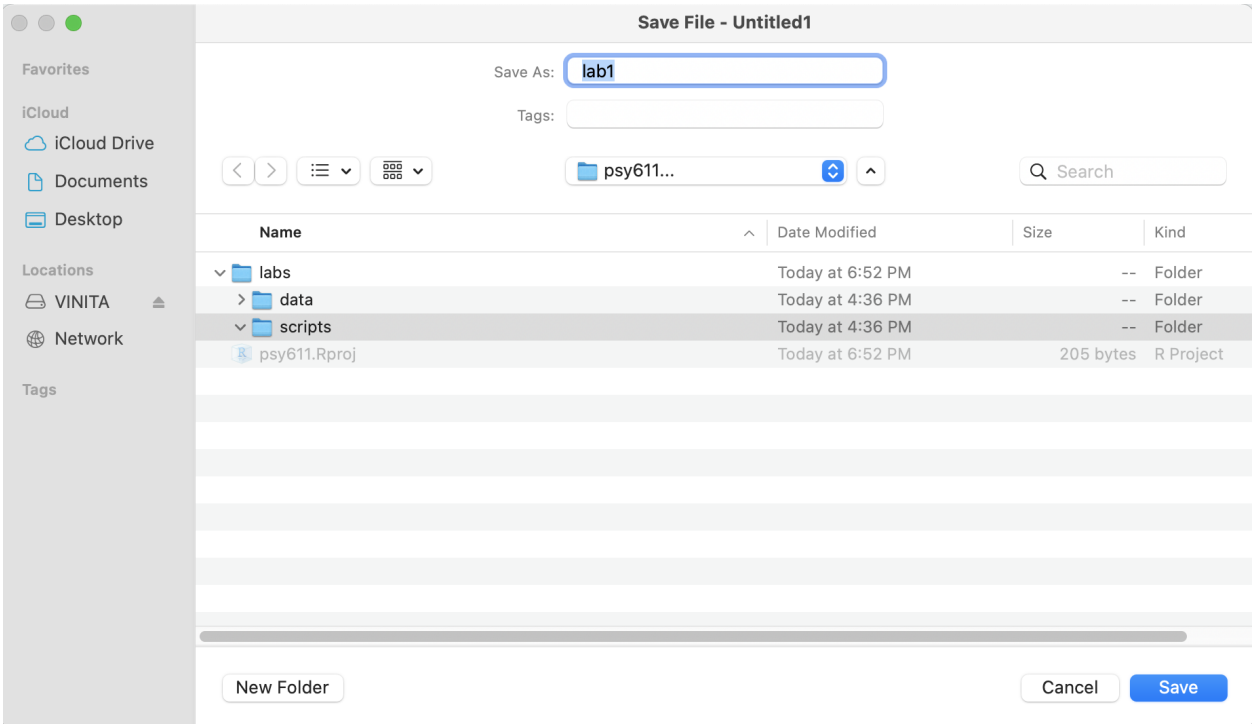
1. Click on the blank piece of paper with the plus sign over it in the upper left-hand corner of RStudio.
2. Click on `R Markdown...`.



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



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

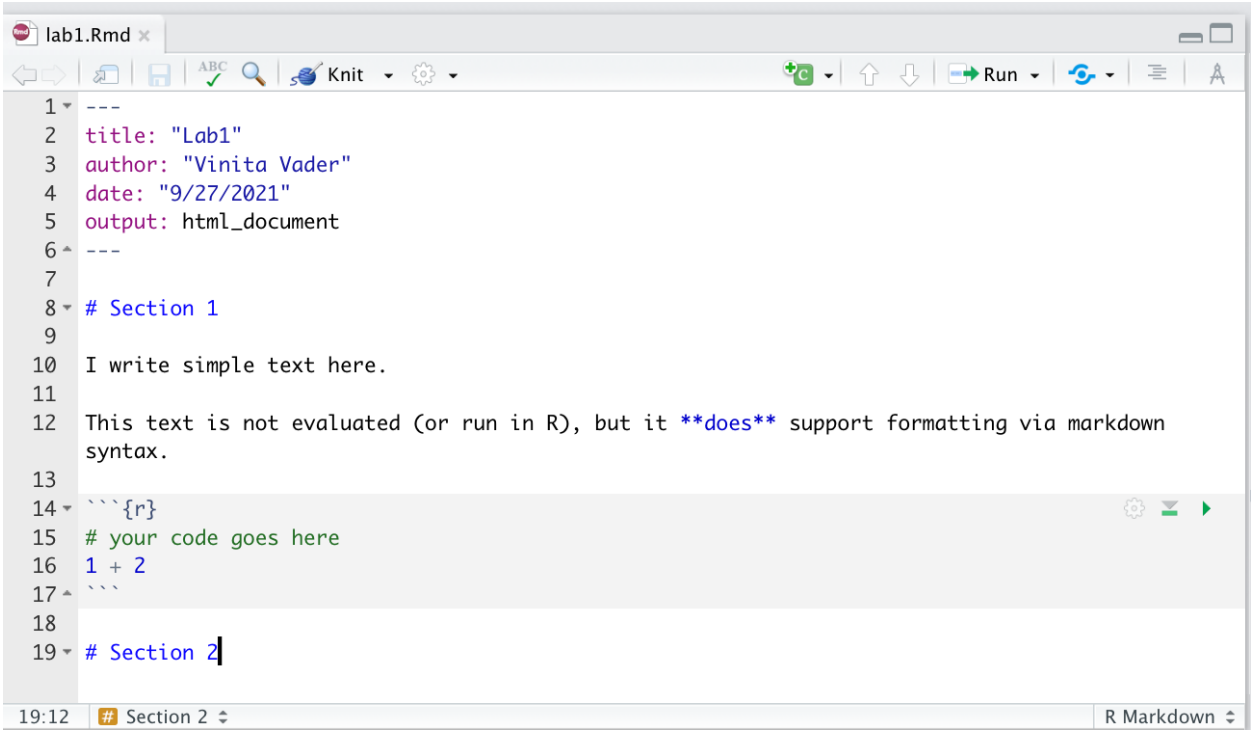


Using an R Markdown Document

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.

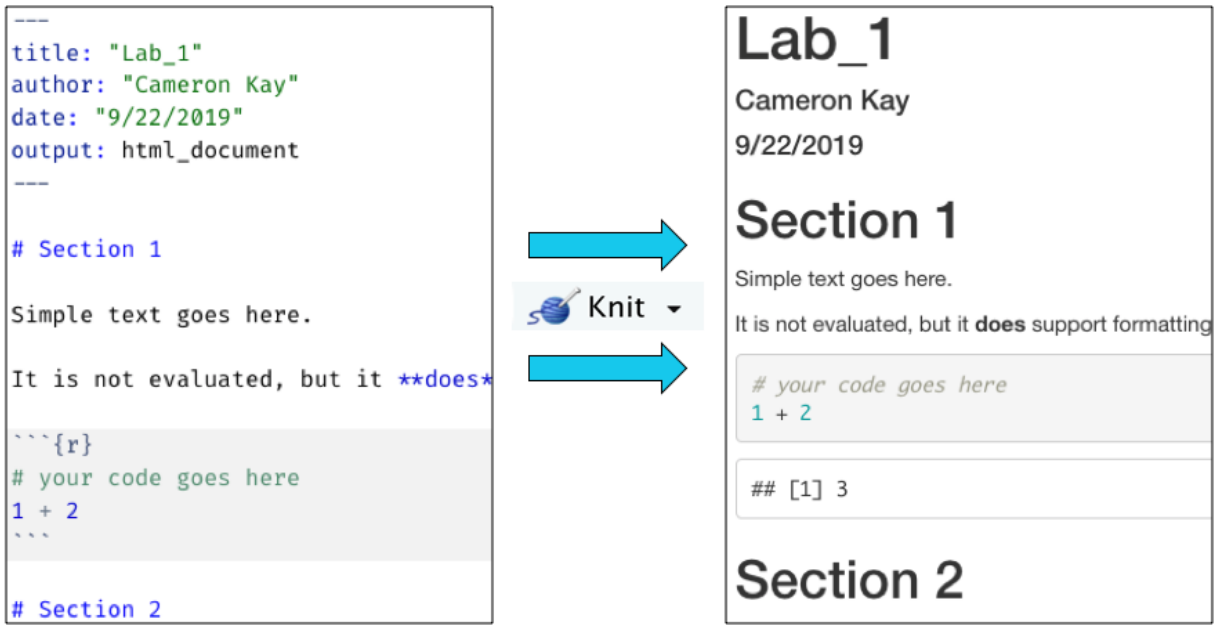
Markdown syntax is used to format the simple text, such as italicizing words by enclosing them in asterisks (e.g., `*this is italicized*` becomes *this is italicized*) or bolding words by enclosing them in double-asterisks (e.g., `**this is bold**` becomes **this is bold**). For a quick rundown of what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

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`.

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 (`^`). You can try running the code below.

```
10 + 5
10 - 5
10 * 5
10 / 5
10 ^ 5
```

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
 10 + 5 * 2
```

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
```

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
```

Now try running the code below.

```
print(my_number <- 10)
(my_number <- 10)
```

There are different situations in which these variations in obtaining values from a stored object are helpful.

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
```

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
```

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
```

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

```
rm(my_number)
```

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"`

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, 30, 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, 30, 40, 50, 60)
```

To print out the entire vector, we can type `my_vector` into the console.

```
my_vector
```

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].

```
my_vector[3]
```

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
```

As with single-value objects we can perform arithmetic operations on vectors, but the behaviour 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
```

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
```

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
```

---

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)
```

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. Lists are helpful in data cleaning and manipulation.

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.

```
mtcars
```

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]
```

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"]
```

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]
mtcars[, "gear"]
```

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, ]
mtcars["Datsun 710", ]
```

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

```
mtcars$gear
```

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")]
```

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))
```

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", "cond_a"))
```

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))
```

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))
```

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)
typeof(10)
typeof("hello")
typeof(TRUE)
```

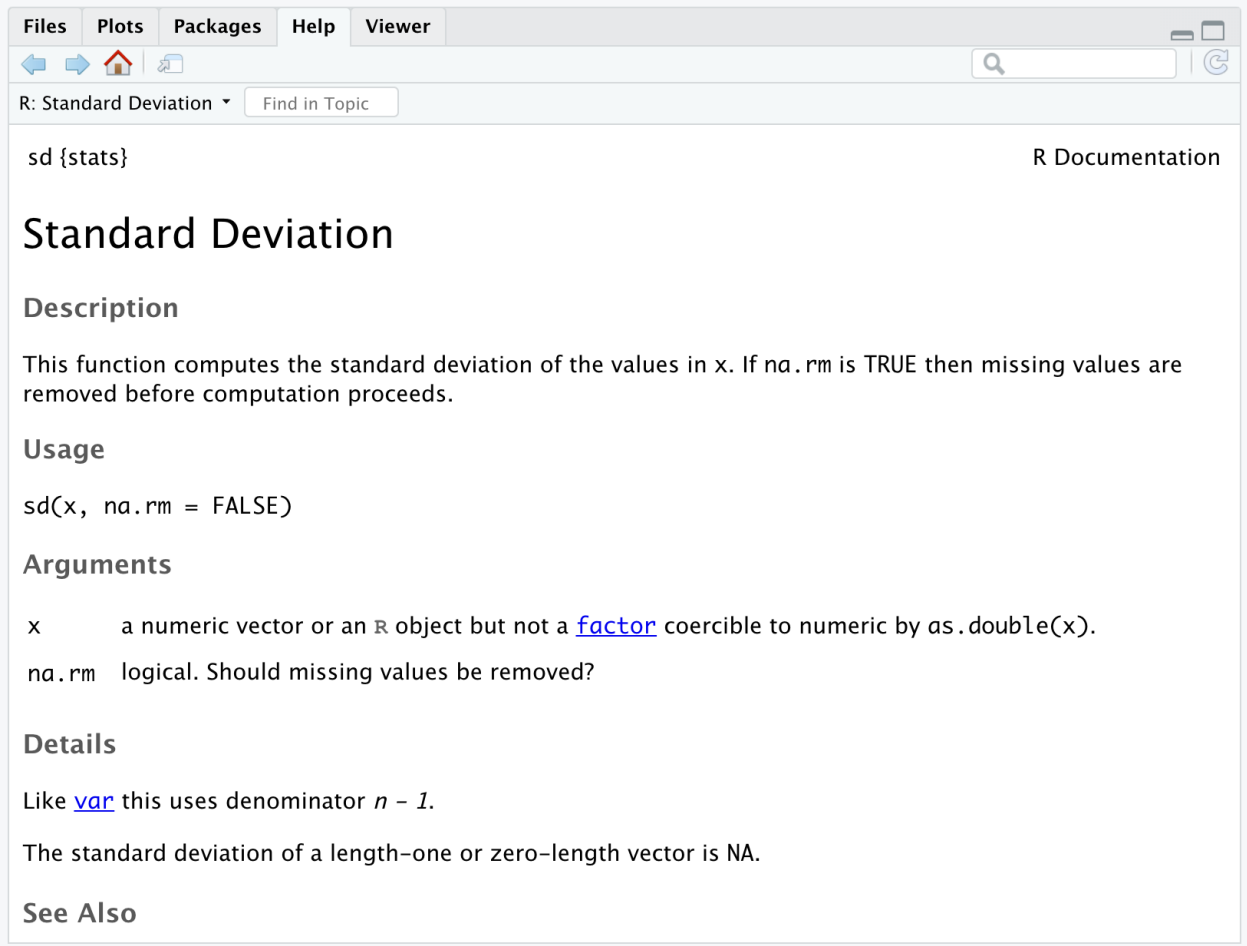
Using the suite of `as.*()` functions (e.g., `as.numeric()`, `as.character()`, `as.logical()`, `as.integer()`), we can likewise coerce objects to other types.

```
as.numeric("10")
as.character(10)
as.logical(1)
as.integer(10.3)
```

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.

```
?sd
```



From the help documentation we can see that `sd()` takes two arguments: (1) An R object and (2) a logical value indicating whether `NA`s (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)
```

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

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

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.

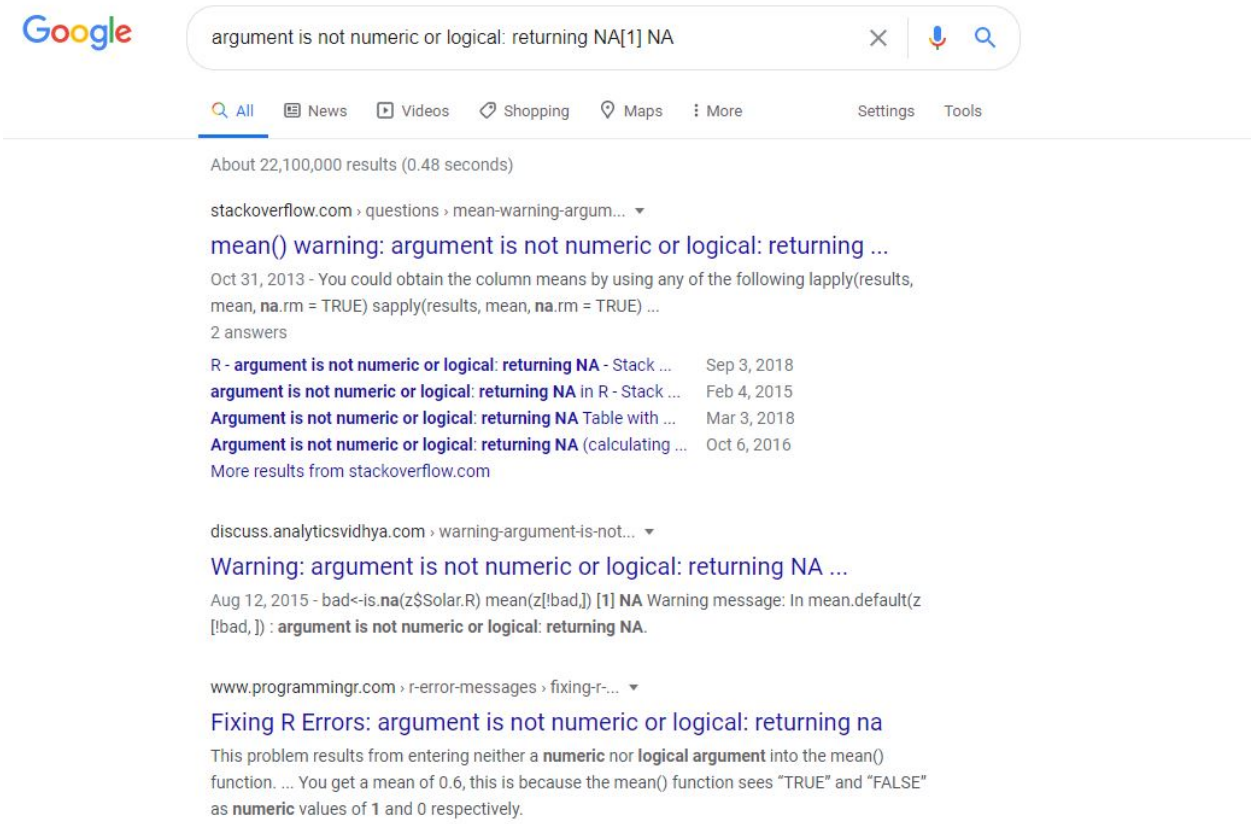
Googling your error message

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.

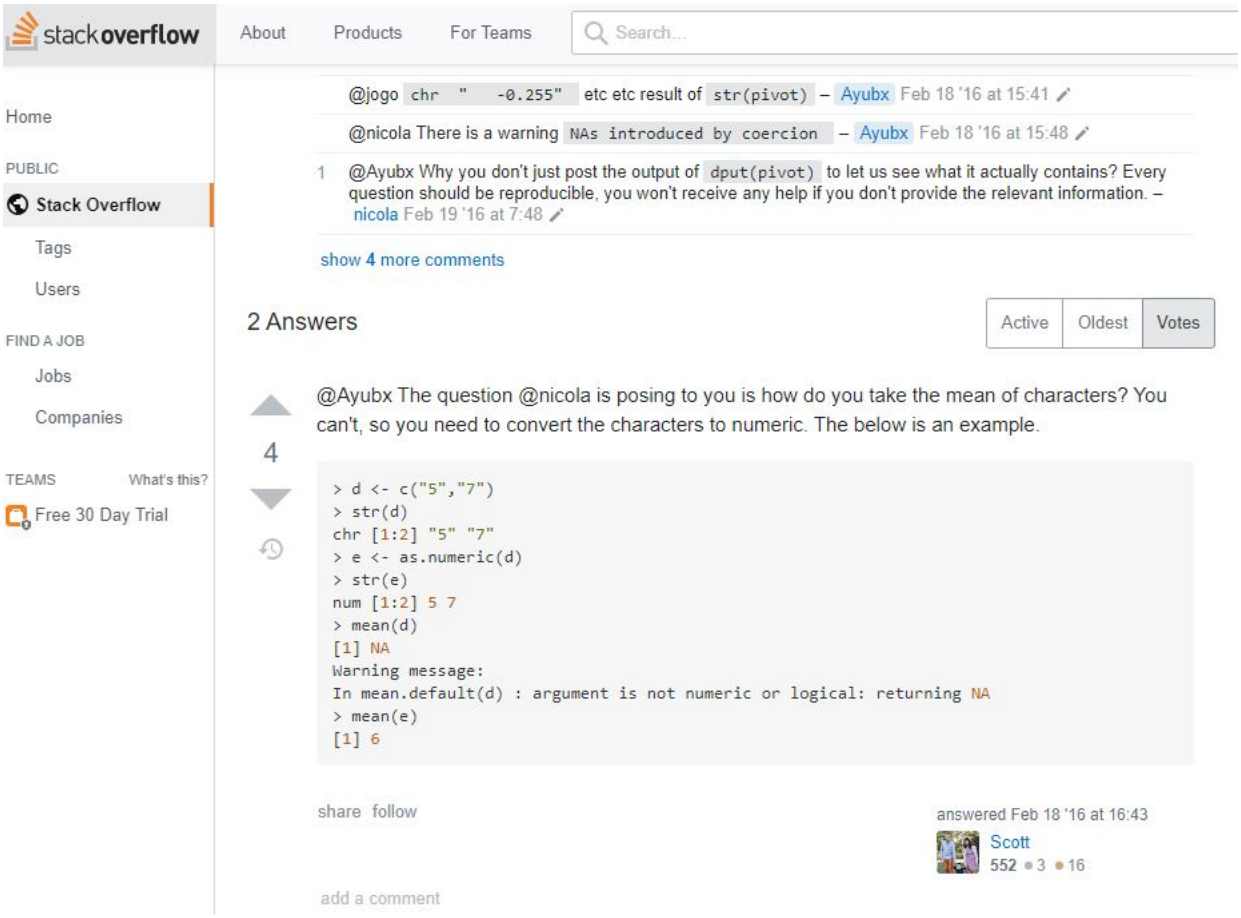
```
mean(c(4,5,"6",7,5))
```

```
> mean(c(4,5,"6",7,5))
[1] NA
Warning message:
In mean.default(c(4, 5, "6", 7, 5)) :
  argument is not numeric or logical: returning 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))
```

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 + 1
 2 + 2
#3 + 3
 4 + 4
```

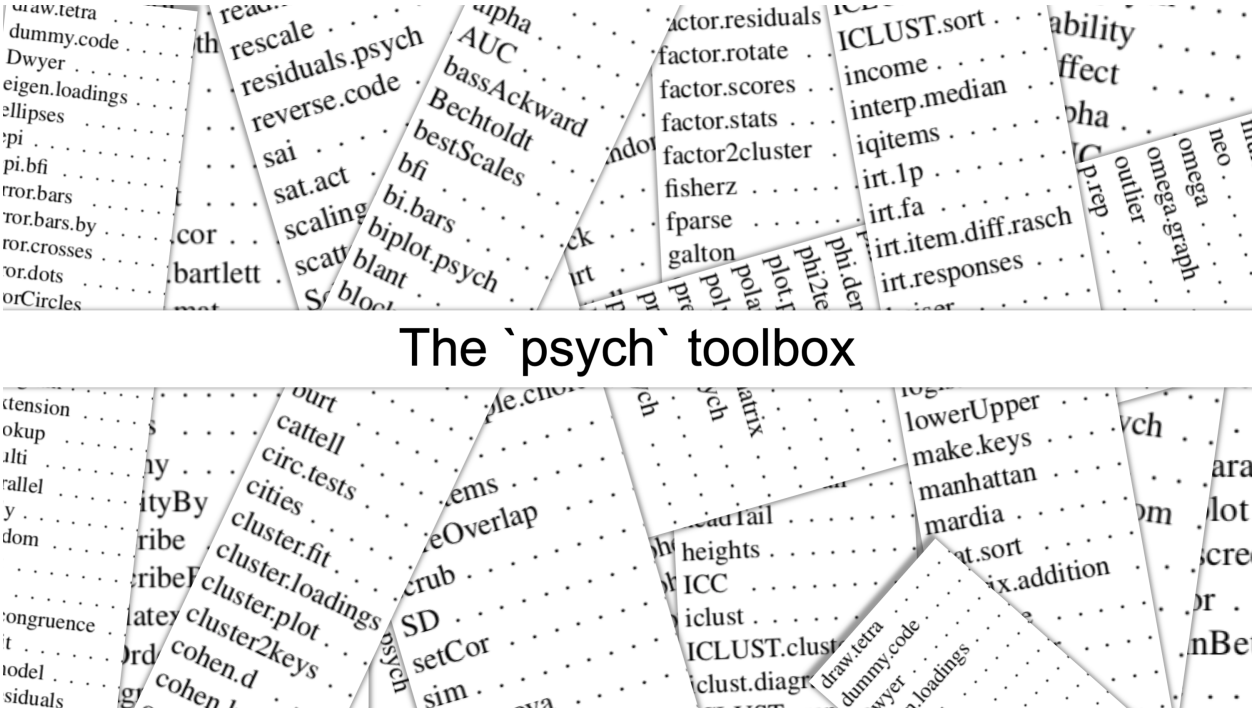
Comments are mostly used to remind yourself (or other people) *what* a piece of code does and *why* the code is written the way that it is. Below is a piece of code that checks if a string is a valid phone number. We can see that the comments explain, not only what each piece of code is doing, but also 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 the country codes because all of the numbers are eith
grepl("^\\(*\\d{3}\\)* ( |-)*\\d{3}( |-)*\\d{4}$", phone_number)
```

Packages

A `package` can include code, documentation for that code, and/or data. A helpful way to think of packages is as a toolbox full of data analysis tools.



There are general purpose toolboxes that contain tools for running common analyses in psychology (e.g., `{psych}`), toolboxes for helping your run advanced statistical models (e.g., `{lavaan}`; `{lmer}`), toolboxes for text mining (e.g., `{tidytext}`), and toolboxes for plotting (e.g., `{ggplot2}`, `{gganimate}`). If you have a problem that needs to be solved, there will probably be a package for it.

Installing packages

To install a package onto your computer, you simply pass the name of the package to `install.packages()`. As a demonstration, we install the `{psych}` package below. The `{psych}` package has several useful data analysis tools for psychologists.

```
install.packages("psych")
```

Note. When installing packages, the package name must be enclosed in quotes: `install.packages("psych")` **NOT** `install.packages(psych)`. You generally only need to install a package once.

Loading a package

Just because we've installed a package to our computer doesn't mean we have access to its functions. Buying a toolbox doesn't necessarily give you access to its tools. You also have to open the toolbox. To open `{psych}` and load its functions, we use `library()`.

```
library(psych)
```

Note. A package can be loaded with or without quotes: `library("psych")` **OR** `library(psych)`. We have to load a package every time that R is restarted.

Try psych commands

Now that we have installed and loaded the {psych} package, let's try out of some its commands.

Using `corr.test()` we can make a correlation matrix of the variables in `mtcars`.

```
corr.test(mtcars)
```

Using `skew()`, we can look at the skew of all of the columns in `mtcars`.

```
skew(mtcars)
```

We can also use `t2d()` to calculate the Cohen's d for a t-value of 3.00 with 300 participants.

```
t2d(t= 3.00, n = 300)
```

This is only a small subset of the functions available in the {psych} package, and {psych} is only one package of over 18,262 on CRAN (as of 27th September, 2021). This is not to mention the tens of thousands of packages hosted on online repositories like GitHub.

Now try evaluating the following code. You will need to install the {yarrr} package. Remember to load the {yarrr} package before you try out of some its commands (adapted from **YaRrr: A Pirate's Guide to R**).

```
#Install and load the package
install.packages("yarrr")
library(yarrr)

# we can look at the documentation for the pirates dataset
?pirates

# we can look at the first six rows of the dataset
head(pirates)

# we can calculate the mean age of the pirates in the dataset
mean(pirates$age)
# the maximum age
max(pirates$age)
# and create a count table of the pirates' genders
table(pirates$sex)

# we can make a scatterplot with the pirates' height on the x-axis and the pirates' weight on the y-axi
plot(pirates$height, pirates$weight)

# we can customize that plot
plot(x = pirates$height, y = pirates$weight,
     main = "Look, Ma! A scatterplot!",
```

```
     xlab = "Height (in cm)",
     ylab = "Weight (in cm)",
     pch = 16,
     col = gray(.0, .1))

# we can make violin plots looking comparing the ages of the pirates across their swords of choice
pirateplot(formula = age ~ sword.type,
           data = pirates,
           main = "Pirateplot of age by favorite sword")

# we can run a t-test comparing the mean age of pirates with a headband and the mean age of pirates wit
t.test(formula = age ~ headband,
       data = pirates,
       alternative = "two.sided")

# we can construct a linear model predicting the number of tattoos a pirate has from their sword of cho
tat.sword.lm <- lm(formula = tattoos ~ sword.type,
                   data = pirates)

# and we can conducted an analysis of variance looking at whether the number of tattoos differs by thei
anova(tat.sword.lm)
```

Importing Data into R

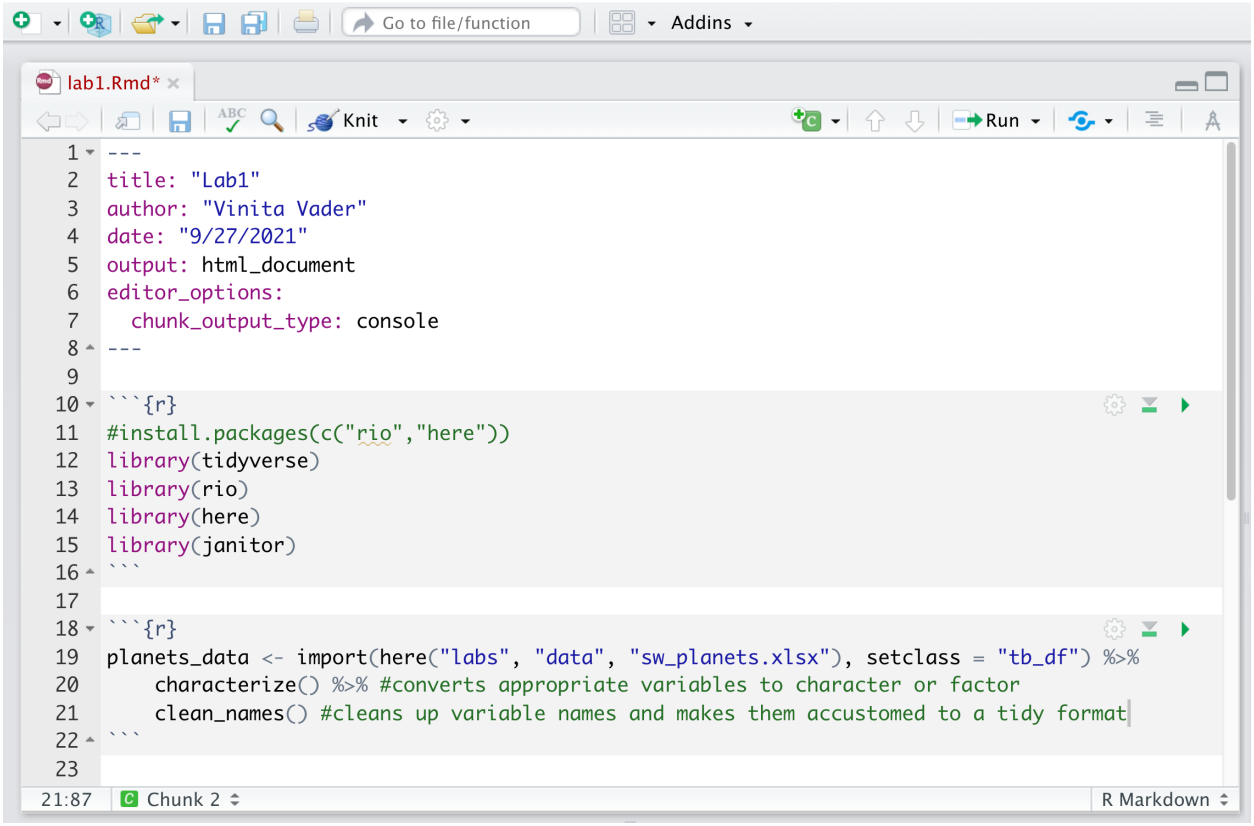
The final topic that we will cover in this lab is how to load data into R. Over the course of your grad school careers (and many times in this class) you will need to import data into R to be analyzed.

For this example, we will be using the planets data set from Star Wars. The data can be downloaded here.

There used to be file-type-specific functions to load data into R (e.g., `read.csv`, `read_excel`) a few years ago. You might see these often if you are looking up solutions for your coding problems. The `{rio}` package streamlines this process by having a single import function (`import()`) that infers the file type from its extension (e.g., `.csv`, `.xlsx`, `.sav`). It also allows you to specify the class and format of your data when you import it; this feature very useful when you are trying to streamline your workflow. Additionally, the `{here}` package makes it very easy to reference folders in your directory.

As we did for `{psych}`, if you don't have these packages already, you will first need to install `{rio}` and `{here}`. You can install `{rio}` and `{here}` with the following code: `install.packages(c("rio", "here"))`.

Second, we will need to load `{rio}` and `{here}` using the library functions. Then, we will import the data by using the `import` function from the `{rio}` package and the `here` function from the `{here}` package. (If you are using a certain library only once, consider using the the `::` for importing a function instead of loading the entire library). In order to use the `here` function, we need to know which folder my dataset is saved in. In this case, the `sw_planets.xlsx` is saved in `labs` -> `data`. Finally, we will save the data into a variable, `planets_data`.




```
#install.packages(c("rio","here", "janitor", "tidyverse"))
library(tidyverse)
library(rio)
library(here)
library(janitor)

#You may need to modify the following code according to yoour workflow
planets_data <- rio::import(here::here("labs", "data", "sw_planets.xlsx"),
    rio::characterize() %>% #converts appropriate variables to character or factor
    janitor::clean_names() #cleans up variable names and makes them accustomed to a tidy format
```

To ensure it was read in properly, we can look at the first six rows of the imported dataset by using the `head()` function.

```
head(planets_data)
```

We can also look at the last six rows by using the `tail()` function.

```
tail(planets_data)
```

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 $\frac{(102+68)\times(3+2)+1250}{50}$ 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 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`.

```
#Your code goes here
```

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).

```
#Your code goes here
```

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`.

```
#Your code goes here
```

5. Install and load the package `ggplot2`.

```
#Your code goes here
```

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")
```