R Programming in Plain English

Taylor Rodgers

2021-04-02

Contents

Chapter 1

About This Book

I never liked most programming books. The same with most programming courses. The reason is that many of them assume you already have a programming background.

That assumption doesn't work well with the R programming language. It's unique because it's a *statistical* programming language. That means the people often needing to learn it are researchers, statisticians, and data analysts – not computer engineers.

That's why I wrote this book. I wanted to explain R programming to people who don't know how to program. I want to strip away all the bloated, technical jargon used in most texts and explain R programming "in plain English."

It's hard enough to learn statistics. The same for psychology, business, sociology, medicine, economics, marketing, or whatever you were crazy enough to devote the best years of your life to learning. We don't need the programming language we use to analyze our data to be as hard to learn as the subject itself.

1.1 Things You Should Know Before Reading

Since R is used mostly for statistics, you should know some statistics before reading this book. For example, I assume you know what a p-value is. The same for confidence intervals, regression analysis, and R-square.

That being said, this book would make a great companion piece to your first statistics class. If your book or professor references code to perform certain calculations, this will help you understand what it's doing.

With that out of the way, let's begin!

Chapter 2

What is R Programming?

Before getting setup in R, you need to know what R programming is and what's so special about it.

R is a programming language built by statisticians for statistical analysis. It is well-suited towards running advanced analysis, building predictive models, machine learning, and even more routine data management.

R is **open source**. That means it's free. The people who developed it wanted to make it accessible to anyone. Since then, R has moved beyond its humble beginnings and developed into a full-blown community of developers.

Developers within the community contribute to the R language by developing packages. **Packages** contain functions that allow you to perform certain tasks.

What does that mean in plain English?

Imagine what it's like to move into a new apartment. It's usually empty when you first move in, but it has running water, a refrigerator, and place to stay warm. So it meets the basic criteria for living - food, water, and shelter.

That's similar to what **base R** provides. It meets the bare necessities to get started with statistical programming.

An R **package** is like the new furniture you bring into the apartment. It's the new coffee table and couch. It's the new coat of paint. All of those things that take an empty apartment and changes it to meet *your* needs.

And that's what's so great about the R programming language. It has a lot of flexibility. If you find one package confusing or challenging, you can probably find an easier one to perform the same task. Or you can build your own functions instead. The only thing is you have to understand the basics first.

One of those basics is **objects**. We'll cover this extensively in another chapter, but R is an *object-oriented* programming language. That means to succeed with

it, you have to understand the object types and how they're used.

If that sounds overwhelming, don't worry. It's easier than you think. And focusing on these core components will go a long way to helping you succeed in R. As a matter of fact, I'd "guesstimate" that understanding R objects and packages will take care of 80% of your needs with this language.

2.1 Is R the Best Statistical Programming Language?

That's a matter of debate and boy do people debate it. The nice thing about R is that it's free and open source and has a vibrant community that contributes to it. It's also meant for statistics. So much of the base functionality is well-suited towards statisticians and researchers alike.

The two big competitors to R are SAS and Python.

SAS is a statistical programming language built by a for-profit company. It's the opposite of open source, which means you have to pay money to use it. That's not necessarily a bad thing though. It means there's more rigorous quality testing because that's what people expect from software that costs money. R packages, on the other hand, may not be as rigorously quality checked. At least not those developed by the community and not the original developers.

The SAS Institute (the people who make SAS) have a lot of market power. They've been around for far longer and have ingrained themselves in academia and pharmaceutical research. Even though R and Python are the preferred choices for most newer companies and younger professionals, SAS has the advantage of being **legacy code**.

You're probably not familiar with that term if you haven't ever programmed before. What that means is that even if R and Python were better alternatives to SAS, it would cost an organization a lot of money and time to have its programmers go back and re-write all their automated, enterprise-wide code.

Personally, I'm actually a fan of SAS. I took it in my grad program and enjoyed its clean structure. Unlike R, which has so many packages and various ways to do the same things, SAS has the benefit of simplicity and clean documentation. That's nice when you are already forcing yourself to learn statistics and don't want to further stress yourself figuring out a programming language.

Python is an interesting competitor of R. It's interesting because it wasn't built for statistical programming. Like R, it's open source and that has led to data scientists building packages that make statistical programming possible.

Python is probably more popular with data scientists than researchers and statisticians. Since data scientists are often joined at the hip with computer programmers, it naturally lends itself more to that work.

I once worked with a company that ran all of its ETL (extract, transform, and load) processes off of Python. Since the data engineers and architects already used Python for that purpose, it wasn't difficult for them to learn to use it for data science as well.

Is Python better than R? I don't know because I haven't programmed much in Python. In my opinion, R is easier to learn for people who've already programmed in SQL, since it has a more direct syntax. Other people say that Python is easier to learn for programming newbies because it's written more like the English language. I didn't find that to be the case, myself.

If you want to develop automated, enterprise level solutions, I do think Python is a better choice. Mostly because it makes it easier for computer engineers to work with you. But if your goal is statistical analysis and reproducible research, R is better in my mind.

2.2 Things to Remember

- R programming is a free and allows for advanced statistical programming
- R is managed by a community of developers, who contribute to its growth through packages
- You must understand object types to succeed with R

Chapter 3

Getting Setup in R and RStudio

This chapter will tell you how to download **R** and **RStudio**. It will also explain the differences between these two tools and how to navigate their user interfaces.

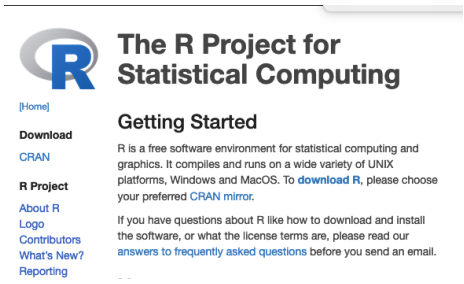
If you have already setup these two tools, feel free to skip ahead to the next chapter. However, if you're still unsure of how to navigate the tools, you may find certain sections useful.

3.1 How to Download R

To download R, go to this website: https://www.r-project.org/

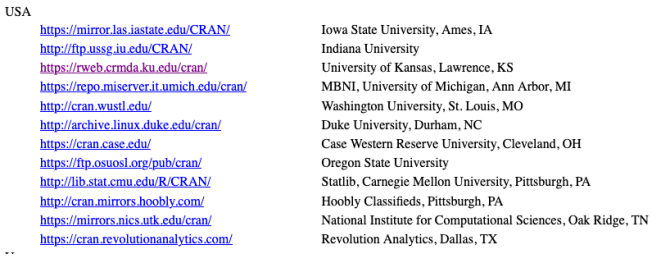
You will then have to go through a series of options and screens to download R. Don't worry. I'll explain each of them along the way.

First, you'll select the "download R" link in the first paragraph.



Second, you'll select a CRAN mirror. Select the one that's closest to you. For example, I live in Lawrence, Kansas. That conveniently has a CRAN mirror in

my own city. If you live in Melbourne, Australia, you'll select the mirror hosted by the University of Melbourne.

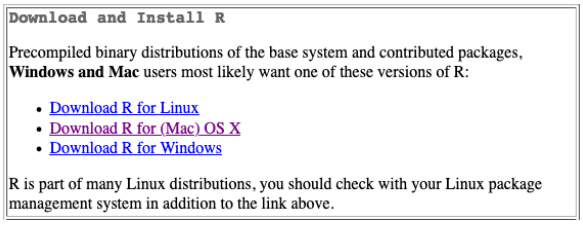


USA

https://mirror.las.iastate.edu/CRAN/	Iowa State University, Ames, IA
http://ftp.ussg.iu.edu/CRAN/	Indiana University
https://rweb.crmda.ku.edu/cran/	University of Kansas, Lawrence, KS
https://repo.miserver.it.umich.edu/cran/	MBNI, University of Michigan, Ann Arbor, MI
http://cran.wustl.edu/	Washington University, St. Louis, MO
http://archive.linux.duke.edu/cran/	Duke University, Durham, NC
https://cran.case.edu/	Case Western Reserve University, Cleveland, OH
https://ftp.osuosl.org/pub/cran/	Oregon State University
http://lib.stat.cmu.edu/R/CRAN/	Statlib, Carnegie Mellon University, Pittsburgh, PA
http://cran.mirrors.hoobly.com/	Hoobly Classifieds, Pittsburgh, PA
https://mirrors.nics.utk.edu/cran/	National Institute for Computational Sciences, Oak Ridge, TN
https://cran.revolutionanalytics.com/	Revolution Analytics, Dallas, TX

Don't worry. There's not much difference to the user in these mirrors. It simply helps optimize your ability to download packages from a nearby source, rather than from some other country in the world. This isn't a big deal for a small time programmer or researcher, but it's beneficial for large scale operations to carefully choose their mirror.

Third, you'll select the option for your operating system:



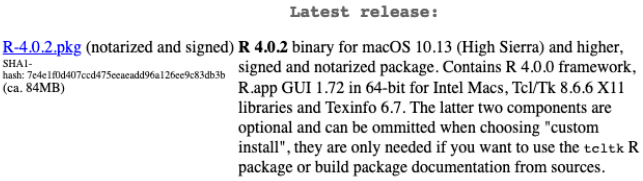
Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

That will take you to a page where you can select the latest release:



Latest release:

R-4.0.2.pkg (notarized and signed) SHA1-hash: 7c4e1f0d407ccd475eeaeadd96a126ee9c83db3b (ca. 84MB)

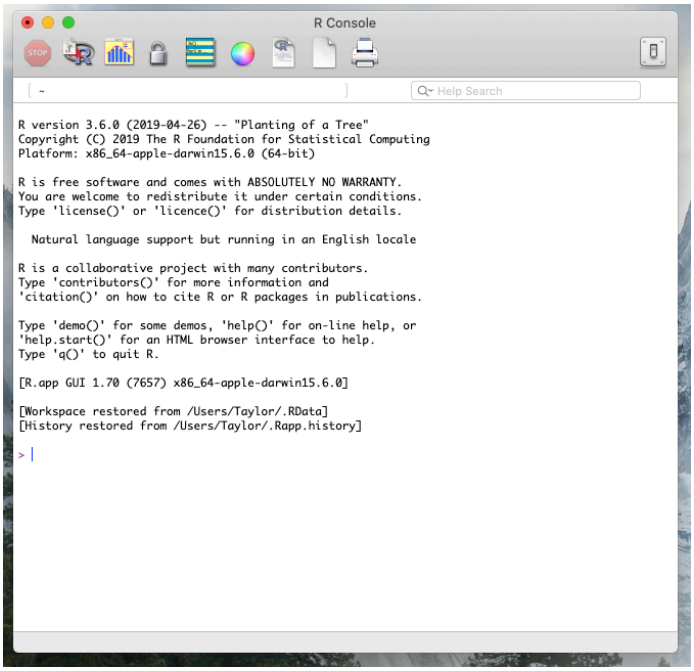
R 4.0.2 binary for macOS 10.13 (High Sierra) and higher, signed and notarized package. Contains R 4.0.0 framework, R.app GUI 1.72 in 64-bit for Intel Macs, Tcl/Tk 8.6.6 X11 libraries and Texinfo 6.7. The latter two components are optional and can be ommitted when choosing "custom install", they are only needed if you want to use the `tcltk` R package or build package documentation from sources.

Download and install the latest release.

3.2 Understanding Base R's User Interface

I won't focus too much on the R interface. We won't use it much in this book. We'll be using RStudio (more on that in a bit). However, go ahead and open up R. If you have a Mac, it will have this icon:

Once you open it, you should see the **R console**. The R console is where you type commands. You'll continue to use the console in RStudio later.



Go ahead and input `2*2` to get a feel for it. You can hit **Ctrl+Enter** on a Windows version and **Command+Enter** on a Mac OS to run the command.

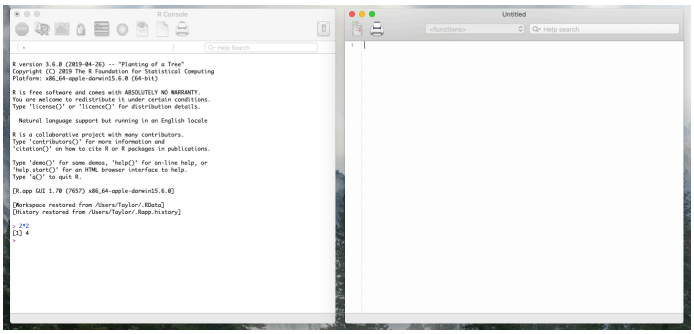
```
2*2
```

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

As you notice, the R console doesn't allow you to go back and edit previously executed commands.

In order to save and edit a script, you'll need to create a document. A document in this context is basically just a text file that saves the code you want to reference later.

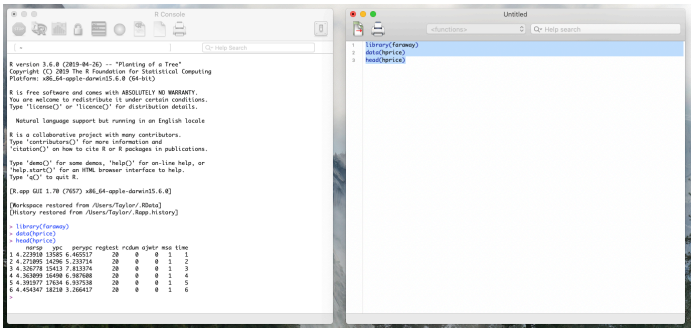
Go to the **File menu** and select **New Document**. You should have two windows afterwards like the screenshot below:



Go ahead and copy and paste this code into the document:

```
install.packages("faraway")
library(faraway)
data(hprice)
head(hprice)
```

Highlight different parts of the code and hit **Command-Enter** or **Ctrl-Enter** and see how it interacts with the console. (It may ask you to re-select a CRAN mirror. Go ahead and re-select the one you used to download.)



Saving scripts this way allows you to modify your code and then re-execute it in the console.

We'll go into more detail about how to input and analyze data later.

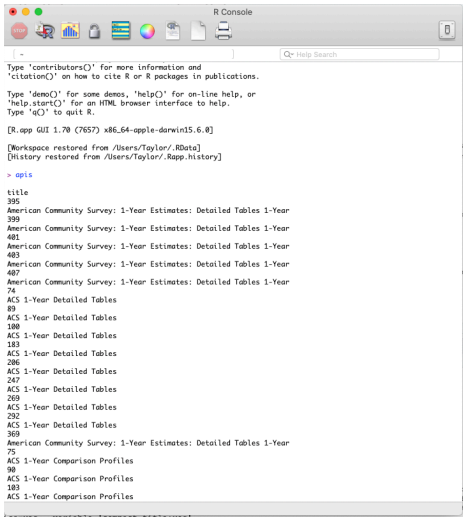
Go ahead and close R for now. No need to save anything.

3.3 Limitations of R Base

The R console in the base version is a simple interface. Sometimes this is nice because you don't get overwhelmed with all the information displayed in RStudio, which we'll cover in a second. If you need to run a few calculations and know your data set well, this is a good setup.

The downside is that the console can't display large amounts of data.

If you look at the screen shot below, I attempted to review metadata from the US Census Bureau. As you can see, it's all squished together because the R console can't cleanly display it.



In order to use the R console for further analysis, I'd have to find a different way to view this data I created.

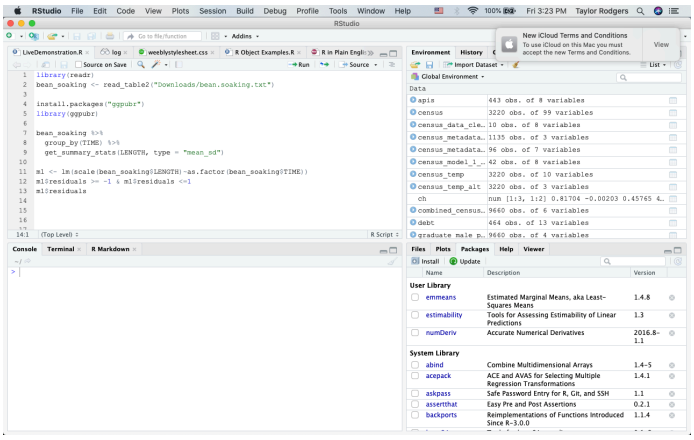
That's where RStudio comes in handy.

3.4 What is RStudio?

If you don't already feel totally confused and overwhelmed with understanding R and R packages – don't worry. I'm about to make things even more confusing by introducing you to RStudio!

RStudio is an IDE (integrated development environment) that allows more interactivity and for you to visually keep track of what you're doing.

In simpler words, it's a handy user interface for programming in R.



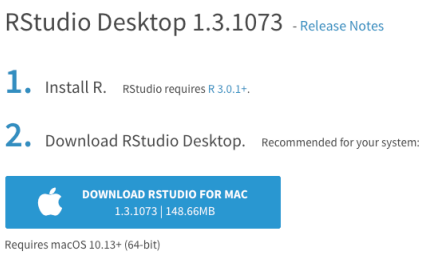
It's far easier to both get started and understand the R programming language by using RStudio. It makes importing data and packages easier. It also makes it easier to manage and visually review what data and packages you have loaded. And lastly, it's just plain nicer looking.

This book will primarily use RStudio for examples. I suggest downloading it to get the best use of my material.

3.5 How to Download RStudio

To download RStudio, go to this website: https://rstudio.com/products/rstudio/download/

Scroll down and you'll see a big blue button to download the latest version of RStudio.



You can tell that RStudio was designed for the end-user in mind by how painless it is to download compared to base R.

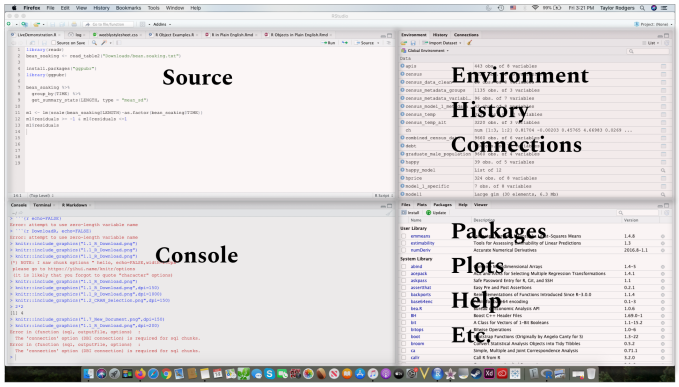
Go ahead and install it once the download finishes.

You'll see an icon like this appear in your applications folder. Go ahead and open it.

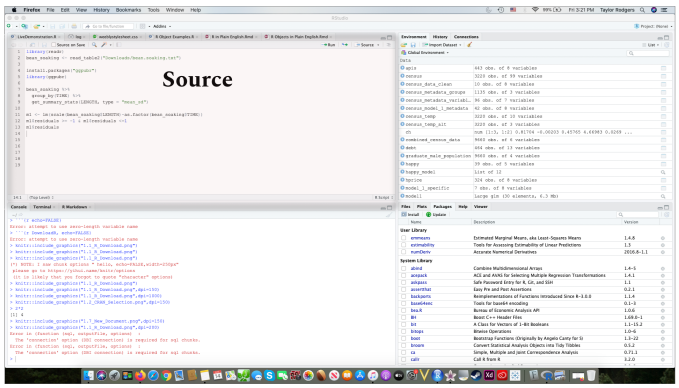
3.6 Understanding RStudio Interface

The RStudio interface is broken up into four panes. The default pane setting has the following:

1. Source
2. Console
3. Environment, History, and Connections
4. Files, Plots, Package, Help, and Viewer

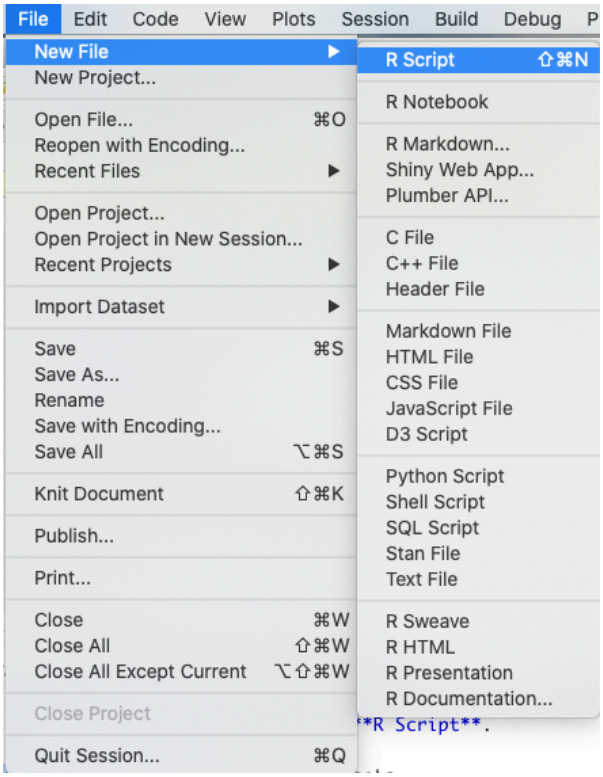


The **source** pane on the top left is a handy one.

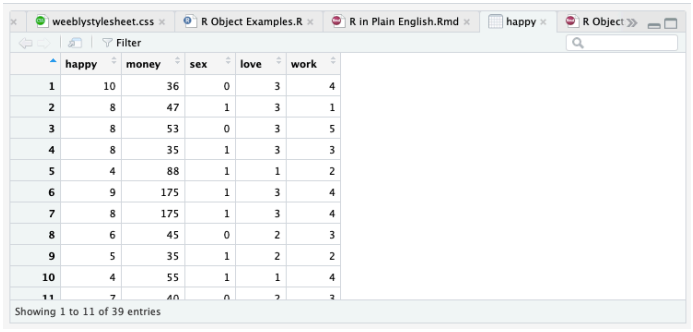


This pane displays previously saved or new R scripts you wrote in the past.

You may not be able to see this pane yet. Especially if you've never opened or saved a script before. To view the source pane, go to the top menu and click **File**, **New File**, and then **R Script**.



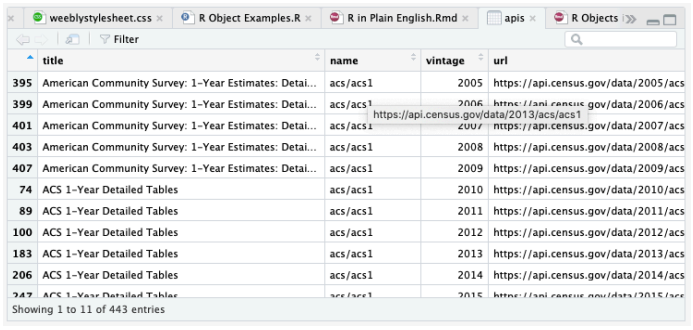
The source pane can also show loaded data sets as well.



	happy	money	sex	love	work
1	10	36	0	3	4
2	8	47	1	3	1
3	8	53	0	3	5
4	8	35	1	3	3
5	4	88	1	1	2
6	9	175	1	3	4
7	8	175	1	3	4
8	6	45	0	2	3
9	5	35	1	2	2
10	4	55	1	1	4
11	7	40	0	2	3

Showing 1 to 11 of 39 entries

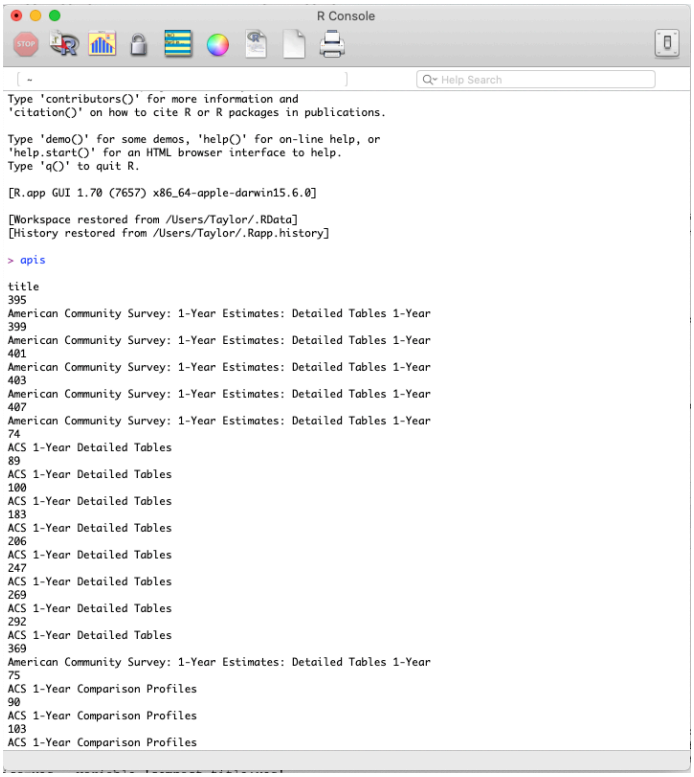
If you recall, I said base R had a limitation if you want to review extremely large data sets. This is where RStudio really comes in handy. Down below is the same Census data set that I attempted to view earlier:



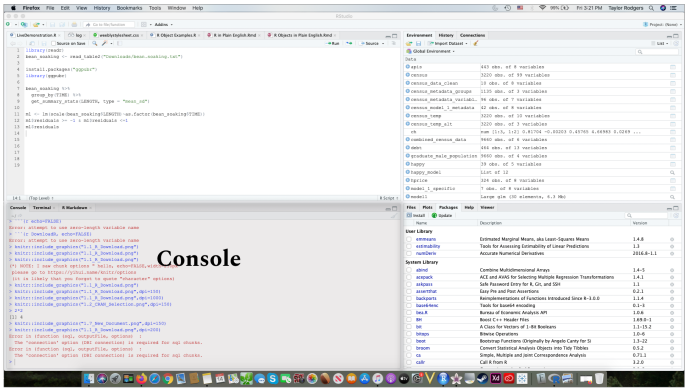
	title	name	vintage	url
395	American Community Survey: 1-Year Estimates: Detai...	acs/acs1	2005	https://api.census.gov/data/2005/acs
399	American Community Survey: 1-Year Estimates: Detai...	acs/acs1	2006	https://api.census.gov/data/2006/acs
401	American Community Survey: 1-Year Estimates: Detai...	acs/acs1	2007	https://api.census.gov/data/2007/acs
403	American Community Survey: 1-Year Estimates: Detai...	acs/acs1	2008	https://api.census.gov/data/2008/acs
407	American Community Survey: 1-Year Estimates: Detai...	acs/acs1	2009	https://api.census.gov/data/2009/acs
74	ACS 1-Year Detailed Tables	acs/acs1	2010	https://api.census.gov/data/2010/acs
89	ACS 1-Year Detailed Tables	acs/acs1	2011	https://api.census.gov/data/2011/acs
100	ACS 1-Year Detailed Tables	acs/acs1	2012	https://api.census.gov/data/2012/acs
183	ACS 1-Year Detailed Tables	acs/acs1	2013	https://api.census.gov/data/2013/acs
206	ACS 1-Year Detailed Tables	acs/acs1	2014	https://api.census.gov/data/2014/acs
247	ACS 1-Year Detailed Tables	acs/acs1	2015	https://api.census.gov/data/2015/acs

Showing 1 to 11 of 443 entries

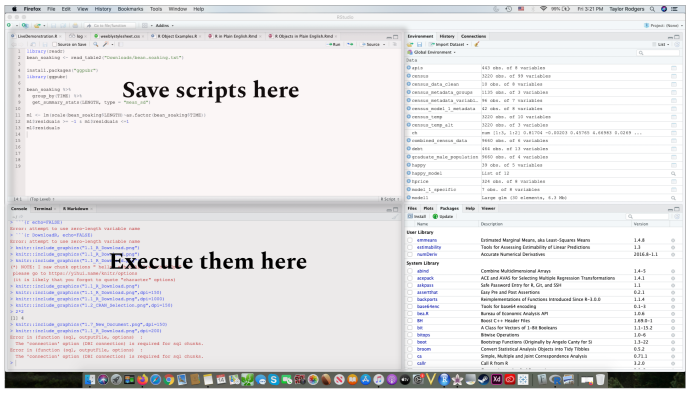
And here is how it looked in base R:



The **console** pane on the bottom left is more or less the same as the base R console we reviewed earlier. It allows you to enter commands.



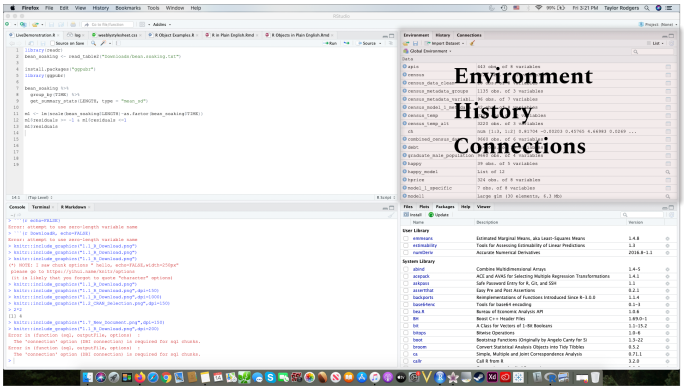
However, you won't be able to save those commands as a script unless you write them in the source pane above. That's similar to what we did with base R and that new document.



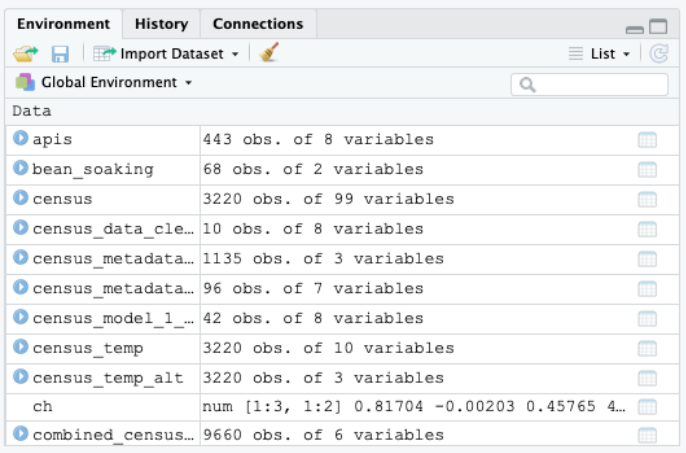
This pane also has a **Terminal** and an **R Markdown** tab (if you have the latter installed). The **Terminal** allows you to enter commands to interact with your computer. IT professionals use this frequently. You probably won't.

R Markdown may not appear for you yet. I'll explain what R Markdown in a later chapter, but this tab merely shows the log for producing an R Markdown export.

The top right pane includes **environment**, **history**, and **connections** as tabular options.

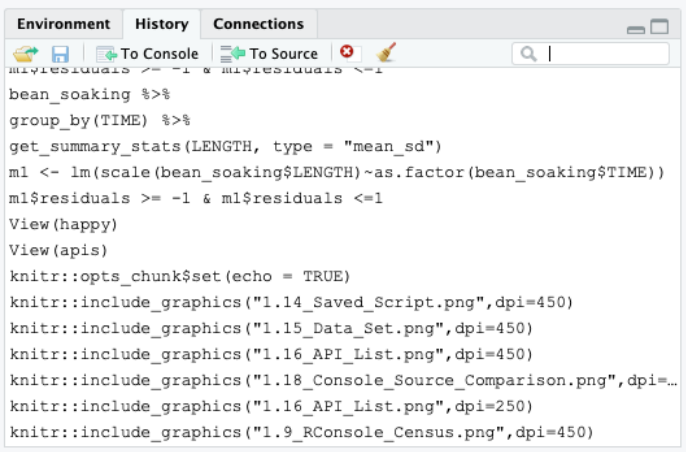


Environment is very handy and it's something that sets RStudio apart from the base version. It shows **objects** with assigned names that are saved in your environment.

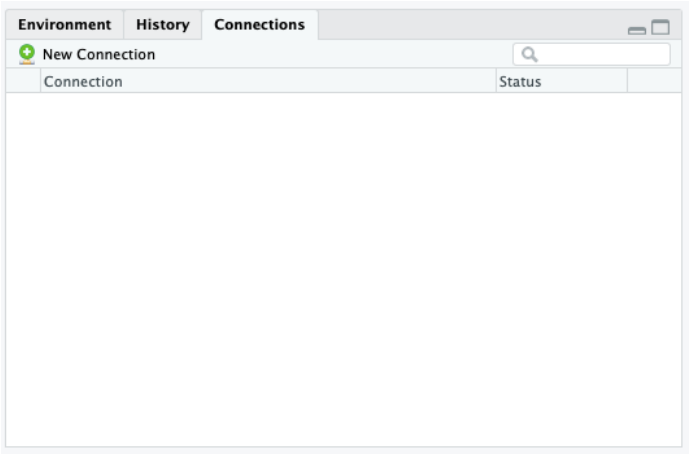


We'll go into objects extensively in the next chapter. Along with packages, *they're the most important component of R programming.* What you need to remember here is that the environment tab in this top right pane tells you what you have saved.

The **history** section is one I don't use too often, but I could see why some people would find it handy. It tells you what commands you've run during your R session. So anything you input in your console will show up here as a record.

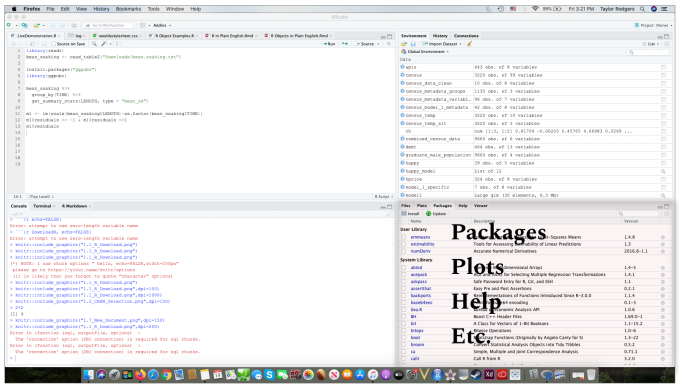


The **connection** tab is useful for those who want to connect to a database or data warehouse.

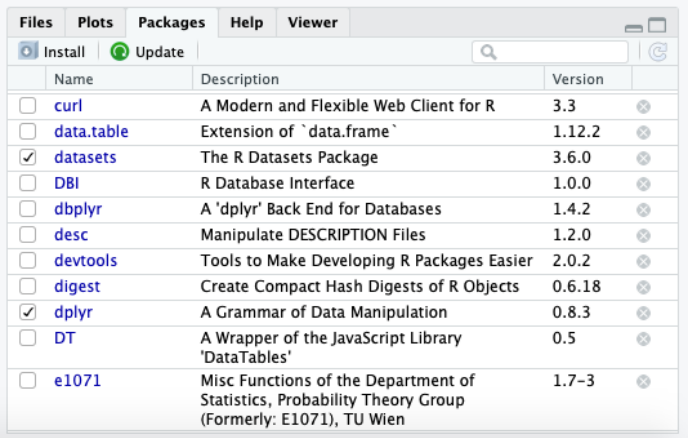


If you're a researcher, you may not use this much. If you work with databases in any capacity, this will make it easier to simply query data directly from the database, as opposed to importing it in via CSV files or spreadsheets.

The bottom right pane is a very useful addition provided by RStudio. It contains a separate tab for **files**, **plots**, **packages**, **help**, and **viewer**.



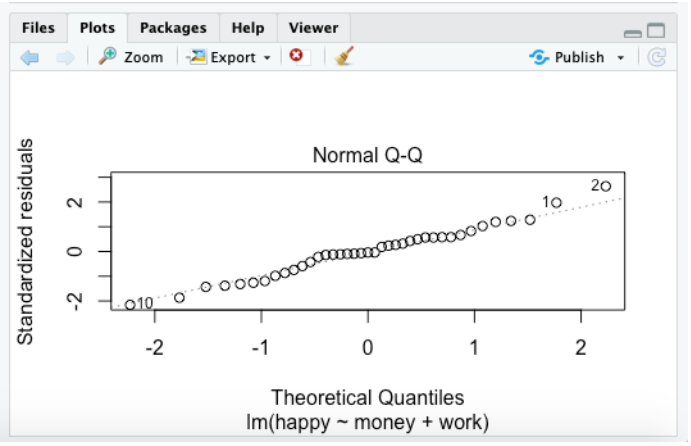
I'm going to start with the **packages** tab. If you recall, I said that packages are what makes R such a useful programming language. It allows you to customize and import functions to suit your needs.



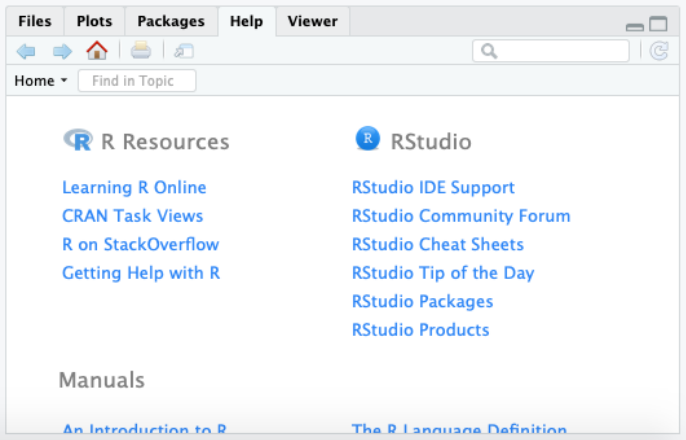
This is a useful little tab. If you open a previously saved script, you may find out that some functions need a package to execute. This tab can tell you whether you need to install that package or simply need to re-load it.

I'll have a whole chapter dedicated to finding packages, installing packages, and loading packages.

Plots is a tab that displays any plots you create using graphical commands. We'll cover this in more detail in a later chapter.



Help is a super helpful tab. You can find the extensive R documentation there that explains many of the functionality of R and how it operates. It also will display information you look up on packages and functions you download.

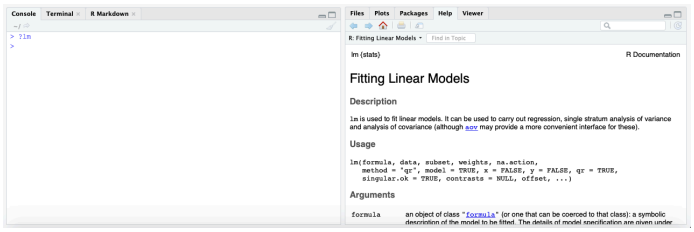


If you ever add a `?` before a function, it will display the documentation in the help section.

Try adding the following command to the R console and see what happens:

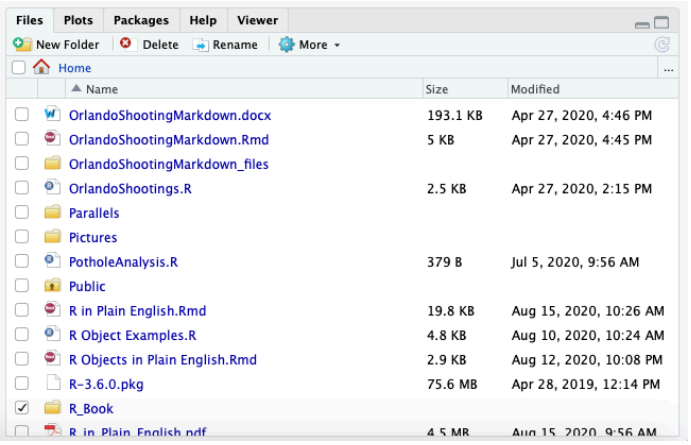
```
?lm
```

Here's what it should look like on your screen:



Don't forget about this trick! It helps a lot!

Lastly, we have the **files** and the **viewer** tab.



The **files** tab displays all the files you can open in RStudio for a given folder. This is helpful because you don't have to specify a file path when loading files listed here.

The **viewer** tab allows to display non-R related outputs, such as a website or JavaScript graphics.

3.7 Things to Remember

- R packages contains new functions that allow you to use R for your own specific purposes
- RStudio is a more user friendly interface

Chapter 4

R Objects Types

R uses **objects** to store and interact with data and there are various object types. That probably means little to you now, but understanding these differences will make R programming easier – whatever your R programming goals.

In fact, I say it's better to understand how these objects interact with one another over memorizing every base function and package out there.

That's different from the approach I took to learning R. When I learned R, I went straight to learning the base functions. You know, the cool stuff that does the regression analysis and confidence intervals and whatnot.

That didn't work out well for me. I was coming from a SQL background and thought data worked in a similar way with R.

Had I started by learning the objects first, I would've saved a lot more time. I would've done less data manipulation in SQL or Excel and made simpler, more scalable R code.

4.1 Why Do Objects Matter?

Almost everything you program in R does one of the following:

- Reads an object
- Modifies an object
- Produces an object
- Calls upon a pre-existing object

The simple code below utilizes five different object types:

```
confint.lm(happy_model)
```

This code for calculating confidence intervals calls upon a base **function**, evaluates an existing **list**, creates several **vectors** and an **array** to perform the analysis, and then outputs a **matrix**. All five are objects. (You can see the function's script by pasting `stats::confint.lm` in your console.)

Understanding this will help you understand how R can seem to "guess" what it's supposed to do based on the data inputs.

4.2 Understanding Object Types Makes It Easier to Transform and Analyze

Pulling data from one object type is different than pulling data from another. This makes it confusing for people who learned about data through SQL, as opposed to other programming languages.

For example, the following code will select most data types in SQL:

```
SELECT
  Field1,
  Field2,
  Field3,
  Field4
FROM
  Data_Set
```

That's different from R. Data selection in R depends on the object type.

For example, using the command `[6]` next to the object name will select a single value from a **vector** object…

```
money[6]
```

```
## [1] 175
```

But that won't work for a **list** object below…

```
happy_model[6]
```

```
## $assign
## [1] 0 1 2
```

To learn how to select, transform, and analyze data in R requires that you learn the underlying structure first. Once you do that, everything else makes more sense.

4.3 The Basic Objects to Remember

Down below are the common objects in R:

1. Vectors
2. Matrices / Arrays
3. Data Frames
4. Lists
5. Factors
6. Functions

We won't talk about functions in this chapter since they need their own chapter to explain how they work.

4.4 Vectors

Vector is the most basic object within R and there are seven "modes" of vectors: logical, numeric, integer, complex, character, date, and raw.

If that seems to be a lot to remember, don't worry. I'd focus on remembering **logical**, **numeric**, and **character** right now. Those are the ones you'll use most often. Others we'll cover as needed.

Vectors can only be one mode at a time. What that means in plain English is that R can't have a word and a number in the same vector.

You can use the code below to create and view a **logical** vector:

```
v1 <- c(TRUE,FALSE,TRUE)
v1
```

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

What this code does is create a vector using the `c(input, input)` notation. It then assigns the vector the name **v1** using the `<-` notation. (A shortcut to the `<-` command is **Option+"-"**.)

You build a **character** vector in the same way, only that you use `c("input", "input")` notation instead:

```
v2 <- c("Hola","Howdy","Hello")
v2
```

```
## [1] "Hola"  "Howdy" "Hello"
```

And a **numeric** vector (like the name suggests) looks like this:

```
v3 <- c(1:3)
v3
```

```
## [1] 1 2 3
```

The code above used the `c(n1:n2)` notation to create a range of values from n1 to n2, where n1 is 1 and n2 is 4. You can also use notations such as `c(n1, n2, n3, n4)` or like `c(n1:n4, n5:n6)`.

Play around with the code below and see what kind numeric vectors you can make!

```
v4 <- c(4:6,1:7)
v4
```

```
##  [1] 4 5 6 1 2 3 4 5 6 7
```

```
v5 <- c(1,5,5,2,1,4)
v5
```

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

I said before that vectors can only be one "mode" or data type at a time. What that means is that if you attempt to mix numbers or a logical value with a character, it simply changes all values to a character.

The code down below takes our previously made vectors, one a numeric and the other a character, and combines them into a single vector. As you can see by the quotation " " marks around the output, it's changed all the numeric values into characters.

```
v6 <- c(v2,v3)
v6
```

```
## [1] "Hola"  "Howdy" "Hello" "1"     "2"     "3"
```

The vector seems basic and not at all like the data sets you'll be using. That might make you ask – will I even use vectors?

Yes. Yes, you will.

The more complex object, data frame, is comprised of individual vectors. (We'll cover more about the data frame object later).

Many functions will also **output** data in vector form or produce a list composed of vectors.

Vectors are also useful as **inputs** into other functions, as well. If you look below, I used a function from the `censusapi` package. I create a vector beforehand and then use it as an input for the function below it.

```
variable_list <-
    c("B15001_003E","B15001_004E","B15001_005E",
      "B15001_044E","B15001_045E","B15001_046E")
getCensus(name="acs/acs5",
          vintage="2018",
          vars=c("NAME",variable_list),
          region="state:*")
```

You can use vectors in this way on a larger scale. For example, if you write a long script with many functions and references, vectors allow you to create a set of parameters at the beginning.

4.5 Matrices and Arrays

Matrices and arrays in R are multi-dimensional vectors. Matrices have multiple rows and columns. Arrays have two or more dimensions. (More on that distinction in a bit).

Like their vector counterpart, all matrix / array values must be the same mode or data type – not a mix. That means if you can't have a numeric value alongside a character value.

Here's an example of a matrix:

```
matrix1 <- matrix(c(2,0,1,3),nrow=2,ncol=2)
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

Why would R programmers want this? It comes back to the R's use as a statistical programming language. For example, multiple linear regression often has combined variables, which involves multiplying two matrices together.

Providing these two object types in R that are solely for numeric values makes this easier.

For example, you can multiply these two matrices together with the `%*%` command and get the same results you would by using matrix algebra:

```
matrix1 <- matrix(c(2,0,1,3),nrow=2,ncol=2)
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

```
matrix2 <- matrix(c(5,7),nrow=2)
matrix2
```

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

```
matrix1 %*% matrix2 #multiplication
```

```
##      [,1]
## [1,]   17
## [2,]   21
```

That's the same as if you did it yourself by hand.

$$\begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 7 \end{bmatrix} = \begin{bmatrix} 17 \\ 21 \end{bmatrix}$$

Unless you're building the functions that calculate this, I doubt you'll use matrices or arrays all that much. However, it's handy to know what they are and how they can be used. They're often the output of functions as well.

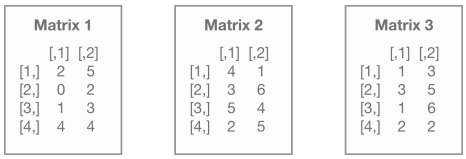
Arrays are more complex than their simple 2-D counterpart. Instead of a single set of rows and columns, you'll have multiple dimensions added on top.

```
matrix3 <- matrix(c(2,0,1,4,5,2,3,4),nrow=4,ncol=2)
matrix4 <- matrix(c(4,3,5,2,1,6,4,5),nrow=4,ncol=2)
matrix5 <- matrix(c(1,3,1,2,3,5,6,2),nrow=4,ncol=2)
array1 <- array(c(matrix3,matrix4,matrix5),
                dim=c(4,2,3))
array1
```

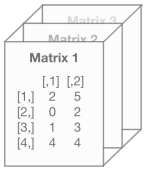
```
## , , 1
##
##      [,1] [,2]
## [1,]    2    5
## [2,]    0    2
## [3,]    1    3
## [4,]    4    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    4    1
## [2,]    3    6
## [3,]    5    4
## [4,]    2    5
##
## , , 3
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    3    5
## [3,]    1    6
## [4,]    2    2
```

If the output up above looks like gibberish, don't worry – most people think it's confusing when they first see it. That's because the R user interface doesn't make it visually intuitive as to how the array operates.

Let's use a picture to visualize this instead. Think of it as if each matrix as a separate entity within the array:



What the array does is simply *stack* them on top of each other:



Since arrays with three dimensions are merely stacked matrices, that means each matrix within the array must have the same number of columns and rows.

4.6 Data Frames

Data frame is the object type that's most similar to what you'd find in a SQL database. What I mean by that is that it's organized and referenced by columns and can have many, many rows.

And most importantly, it can contain both numeric and character values! (Earth shattering, I know!)

I like to think of a data frame as a way to combine vectors. As a matter of fact, you can build a data frame doing just that.

Take this code below and run it. You'll see that we make individual vectors containing data on James Bond movies[1] and we then combine it into a data frame.

```
filmname <-
      c("Skyfall","Thunderball","Goldfinger",
        "Spectre","Live and Let Die",
        "You Only Live Twice",
        "The Spy Who Loved Me","Casino Royale",
        "Moonraker","Diamonds Are Forever",
        "Quantum of Solace","From Russia with Love",
        "Die Another Day","Goldeneye",
        "On Her Majesty's Secret Service",
        "The World is Not Enough",
        "For Your Eyes Only","Tomorrow Never Dies",
        "The Man with the Golden Gun",
        "Dr. No","Octopussy",
        "The Living Daylights","A View to a Kill",
```

[1] ?

```
        "Licence to Kill")
year <- c("2012","1965","1964","2015",
          "1973","1967","1977","2006",
          "1979","1971","2008","1963",
          "2002","1995","1969","1999",
          "1981","1997","1974","1962",
          "1983","1987","1985","1989")
actor <- c("Daniel Craig","Sean Connery",
           "Sean Connery","Daniel Craig",
           "Roger Moore","Sean Connery",
           "Roger Moore","Daniel Craig",
           "Roger Moore","Sean Connery",
           "Daniel Craig","Sean Connery",
           "Pierce Brosnan","Pierce Brosnan",
           "George Lazenby","Pierce Brosnan",
           "Roger Moore","Pierce Brosnan",
           "Roger Moore","Sean Connery",
           "Roger Moore","Timothy Dalton",
           "Roger Moore","Timothy Dalton")
gross <- c(1108561008,1014941117,912257512,
           880669186,825110761,756544419,
           692713752,669789482,655872400,
           648514469,622246378,576277964,
           543639638,529548711,505899782,
           491617153,486468881,478946402,
           448249281,440759072,426244352,
           381088866,321172633,285157191)/1000000
bond <- data.frame(filmname=filmname,
                   year=year,
                   actor=actor,
                   gross=gross)
```

And if you use the $ sign, as we discussed before, you can re-select the individual vectors back out of it.

```
bond$filmname
```

```
##  [1] "Skyfall"                         "Thunderball"
##  [3] "Goldfinger"                      "Spectre"
##  [5] "Live and Let Die"                "You Only Live Twice"
##  [7] "The Spy Who Loved Me"            "Casino Royale"
##  [9] "Moonraker"                       "Diamonds Are Forever"
## [11] "Quantum of Solace"               "From Russia with Love"
## [13] "Die Another Day"                 "Goldeneye"
## [15] "On Her Majesty's Secret Service" "The World is Not Enough"
## [17] "For Your Eyes Only"              "Tomorrow Never Dies"
```

```
## [19] "The Man with the Golden Gun"     "Dr. No"
## [21] "Octopussy"                       "The Living Daylights"
## [23] "A View to a Kill"                "Licence to Kill"
```

If you want to select a single column and maintain the data frame object type, you have to use the following code:

```
bond[1]
```

```
##                            filmname
## 1                           Skyfall
## 2                       Thunderball
## 3                        Goldfinger
## 4                           Spectre
## 5                  Live and Let Die
## 6                You Only Live Twice
## 7              The Spy Who Loved Me
## 8                     Casino Royale
## 9                         Moonraker
## 10             Diamonds Are Forever
## 11                Quantum of Solace
## 12            From Russia with Love
## 13                  Die Another Day
## 14                        Goldeneye
## 15 On Her Majesty's Secret Service
## 16          The World is Not Enough
## 17               For Your Eyes Only
## 18              Tomorrow Never Dies
## 19     The Man with the Golden Gun
## 20                            Dr. No
## 21                        Octopussy
## 22             The Living Daylights
## 23                 A View to a Kill
## 24                  Licence to Kill
```

We'll go into further detail about selecting, transforming, and analyzing data frames later on. The way you go about it depends on whether you want to make efficient code or you want to make "readable" code for other analysts.

4.7 Factors

Factors take vectors (or data frame columns) and create categories to group the values. Confused? It's actually fairly simple.

Think back to the data frame we built for the Bond films. If you use the code below, you'll see the first six rows:

```
head(bond)
```

```
##              filmname year        actor     gross
## 1             Skyfall 2012 Daniel Craig 1108.5610
## 2         Thunderball 1965 Sean Connery 1014.9411
## 3          Goldfinger 1964 Sean Connery  912.2575
## 4             Spectre 2015 Daniel Craig  880.6692
## 5    Live and Let Die 1973  Roger Moore  825.1108
## 6 You Only Live Twice 1967 Sean Connery  756.5444
```

Now let's say you want a short list of the Bond actors. If you'll notice in the data set, the actor names like "Daniel Craig" and "Sean Connery" are used repeatedly. These are basically ways to group the data frame with a common field name - the actor who played Bond.

If we tried to get a list of these actors using the `levels()` function, it wouldn't work.

```
levels(bond$actor)
```

```
## NULL
```

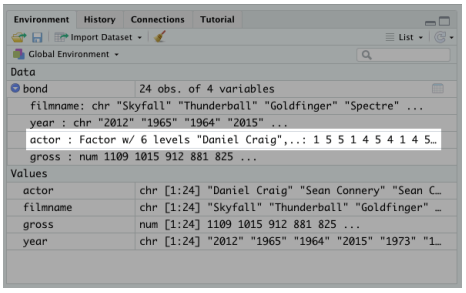
That's because it hasn't been **factored** yet.

This is a real simply fix. Simply use the `factor()` function and assign it to the field name within the data frame. You can use the code below to do this.

```
bond$actor <- factor(bond$actor)
levels(bond$actor)
```

```
## [1] "Daniel Craig"   "George Lazenby" "Pierce Brosnan" "Roger Moore"
## [5] "Sean Connery"   "Timothy Dalton"
```

And this will also show up in the environment tab in the top left.



R used to automatically factor character variables for you. However, that functionality was removed in a recent update.

You may see factors as a not-so-important object type, but that's not true. It comes in handy with regression analysis. Especially if your categorical variables are numeric.

For example, our Bond data frame may not include the actor name. It could simply have a number between 1 and 6 for the actor - with Sean Connery as 1 and Daniel Craig as 6. That means a regression analysis would've analyzed the actor as a continuous variable by default!

This also comes up with experiments that analyze the impact of medicine. It's not uncommon to label one drug as 1 and another drug as 2. That means you'd have to factor those drug codes so that your regression analysis reads them correctly.

4.8 Lists

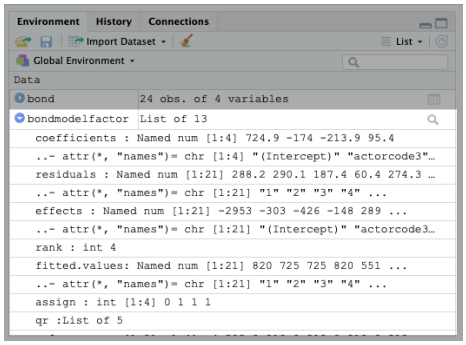
Lists are objects that usually store other objects in a nice bundle. Those objects could be vectors, other lists, data frames, etc.

Many of the more complex R base functions produce lists. A common one is produced by the `lm()` function.

Use the code below to build a model with the James Bond data:

```
bondmodel <- lm(gross~actor,data=bond)
```

Now you can see the list this produces in the environment tab. As you can see, there's a lot in this list.



You can also see what's in the list using the following code:

```
names(bondmodel)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

This gets to the crux of why it's important to know when you're dealing with a list. It changes the way you select components of that list.

For example, let's say you want just the coefficients from a model you had built.

You can use the same $ symbol as before.

```
bondmodel$coefficients
```

```
##         (Intercept) actorGeorge Lazenby actorPierce Brosnan    actorRoger Moore
##           820.31651          -314.41673          -309.37854          -269.48336
##   actorSean Connery actorTimothy Dalton
##           -95.43409          -487.19349
```

However, if you want to select a single coefficient, you have to use a number value afterwards:

```
bondmodel$coefficients[4]
```

```
## actorRoger Moore
##        -269.4834
```

That's why it's important to know if you're pulling from a list or not. It changes the way that you select key parts of the data.

4.9 Functions

Functions are also an object. Most of the time, you'll be using a built-in function that is in the R base code or in a package you loaded.

However, you may find yourself building your own functions, which is handy if you don't want to search for a pre-existing one or need something unique to your situation.

We'll go into more detail about functions in a later chapter because of their complexity.

4.10 Things to Remember

- The key to understanding R is understanding objects
- The object type changes the way you'll read, transform, and product data

Chapter 5

How to Filter and Transform Data in Base R

In the last chapter, I explained the various object types in R. Now we want to learn how to **filter** and **transform** those objects. Notice how I didn't say filter and transform "data"? That's because the methods you use to handle data in R heavily depend on the object type.

Before I explain those methods though, we need to cover operators.

5.1 What Are Operators?

If you're new to programming, then you're probably not familiar with the term **operator**. Operators, in plain English, modify or evaluate data. That's important to data transformation and filtering.

There are two types of operators in R: arithmetic and logical.

Arithmetic operators cover tasks like addition, subtraction, etc. You know? The basic math stuff. This is useful for data transformation and will be used in several examples later. Here are the common arithmetic operators:

label	symbol
addition	+
subtraction	-
multiplication	*
division	/
exponent	^
matrix multiplication	%*%
matrix division	%/%

Logical operators takes the data and generates a TRUE or FALSE output, based on whether the data meets your requirement. This is more helpful for *filtering* data than transforming. Here are the common logical operators:

label	symbol
less than	<
greater than	>
less than or equal	<=
greater than or equal	>=
equal	==
does not equal	!=
and	&
or	|
in	%in%

Don't worry if you're unsure of how to use these just yet. You'll see examples for these in the next few sections. This is just for your easy reference.

5.2 How to Filter and Transform Data From a Vector

Vectors are the easiest object type to filter. Same with transforming the data within them.

If you want to reference or view the entire vector, you simply enter the name you assigned the object:

```
v5 <- c(1,5,5,2,1,4)
v5
```

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

(Remember: the `<-` command allows you to name any object. You can use **Option+"-"** as a short-cut for `<-`.)

You also can select a single entry from a vector using the `[n]` notation:

```
v5 <- c(1,5,5,2,1,4)
v5[3]
```

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

As you can see, the script above selected the third value from the vector.

You can select a range of entries by using the `[n:n]` notation.

```
v5 <- c(1,5,5,2,1,4)
v5[3:4]
```

```
## [1] 5 2
```

And, as we've seen before, you can create a new vector by referencing old vectors!

```
v2 <- c("Hola","Howdy","Hello")
v7 <- c(2:4)
v8 <- c(v2,v7)
v8
```

```
## [1] "Hola"  "Howdy" "Hello" "2"     "3"     "4"
```

You can also use other base R functions to filter data.

For example, you may want to see the minimum or maximum value in a vector. You can use the `max()` and `min()` command to do so.

```
v5
```

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

```
max(v5) # Max value
```

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

```
min(v5) # Min value
```

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

And you can use logical operators as well. In the example below, I use the `>=` and `&` operators to filter values:

```
v5 >= 2 # Values greater than 2
```

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

```
v5 >= 3 & v5 <= 5 # Values between 3 and 5
```

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

We can also use the `|` operator to find values that meet a criteria. For example, I filter the vector below to "Hola" and "Howdy."

```
v2
```

```
## [1] "Hola"  "Howdy" "Hello"
```

```
v2 == "Hola" | v2 == "Howdy"
```

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

You probably noticed that these logical operators only return a TRUE or FALSE statement. That makes sense since it is a *logical* argument that's evaluated. However, we may want to see the actual values that meet the argument. This isn't important in an example like this, but it does come up later on for more complex objects.

To show the actual values where the logical argument is true, you use the `object_name[argument]` notation. In the next few examples, I filter the vectors down to values that meet the arguments used in the last few examples:

```
v5[v5>=2]
```

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

```
v5[v5 >= 3 & v5 <= 5]
```

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

```
v2[v2 == "Hola" | v2 == "Howdy"]
```

```
## [1] "Hola"  "Howdy"
```

In the examples above, I simply took the logical argument and plugged it into the brackets.

You can also change data easily when it comes to numeric vectors. For example, down below is a vector of box office revenue for James Bond films. Copy and paste this script into your R console and execute:

```
gross <-
  c(1108561008,1014941117,912257512,880669186,
    825110761,756544419,692713752,669789482,
    655872400,648514469,622246378,576277964,
    543639638,529548711,505899782,491617153,
    486468881,478946402,448249281,440759072,
    426244352,381088866,321172633,285157191)
gross
```

As you can see, the values are very large. To make our analysis easier, we can use an *arithmetic* operator I showed earlier. In this scenario, I want to make the values smaller. So I'm going to divide it using the `/` operator.

```
gross/1000000
```

You can also calculate individual values this way too.

```
gross[4]
```

```
## [1] 880669186
```

```
gross[4]/100000
```

```
## [1] 8806.692
```

And you can re-assign the value to a particular part of a vector using the methods we described above and the `<-` notation. For example, we can see below how we re-assign values based on the location.

```
v8 <- c(1,5,5,2,1,4) # Creates the original vector
v8[6] <- 8 # Replaces the sixth value with an 8
v8[1:3] <- c(4,3,1) # Replaces the first three values
v8
```

```
## [1] 4 3 1 2 1 8
```

5.3 How to Filter and Transform Data From a Matrix

Filtering the data within a matrix is both similar and different than a vector.

It's similar because we can use the `[n]` notation to select a single entry. We had done this before with a vector:

```
v5[2]
```

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

You can do the same for a matrix. If you run the code below, you'll re-create and view the matrix we used in the last chapter:

```
matrix1 <- matrix(c(2,0,1,3),nrow=2,ncol=2)
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

And here you'll select the fourth value from that matrix using the `[4]` command:

```
matrix1[4]
```

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

Now that isn't very practical for a matrix. You may need to select a value from a specific row or column instead. This is where matrices are different from a vector. You'll want to use the `[r,c]` command to determine which values you want. In the example below, I select the second row and first column of the matrix.

```
matrix1[2,1]
```

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

We can make this easier on ourselves. Instead of specifying row or column numbers, we can give them names. That way, we can use the `[row_name,column_name]` notation to select data from a matrix. Down below, I give our previously created matrix row and column names.

```
colnames(matrix1) <- c("Col1","Col2")
rownames(matrix1) <- c("Row1","Row2")
matrix1["Row2","Col1"]
```

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

We can also apply vector filtering methods to matrices. For example, I want to see what values are greater than 0.

```
matrix1 > 0 # Returns true or false
```

```
##       Col1 Col2
## Row1  TRUE TRUE
## Row2 FALSE TRUE
```

Funny enough though, you can't return the actual values that meet this criteria in a matrix form. It'll turn into a vector. That's because the output may not have the same number of columns and rows as the original matrix. So R assumes it'll need a one-dimensional object output.

```
matrix1[matrix1 > 0]
```

```
## [1] 2 1 3
```

You can use the same techniques we outlined before with the vectors to transform the data within a matrix. Copy and paste the codes below to your R console and see the results. Feel free to play around with the inputs to see what happens.

```
matrix1 <- matrix(c(2,0,1,3),nrow=2,ncol=2)
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

```
matrix1[3] <- 5
matrix1
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    0    3
```

```
matrix1[,2] <- 2
matrix1
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    0    2
```

```
matrix1[2,2] <- 0
matrix1
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    0    0
```

Like the vectors, you can transform the data within the matrix using the arithmetic operators we discussed earlier.

```
matrix1
```

```
##      [,1] [,2]
## [1,]    2    2
## [2,]    0    0
```

```
matrix1 + 2
```

```
##      [,1] [,2]
## [1,]    4    4
## [2,]    2    2
```

```
matrix1 - 4
```

```
##      [,1] [,2]
## [1,]   -2   -2
## [2,]   -4   -4
```

```
matrix1 ^ 3
```

```
##      [,1] [,2]
## [1,]    8    8
## [2,]    0    0
```

```
matrix1 * 5
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]    0    0
```

You can also use these operators to combine matrices. We'll need a few matrices to illustrate these examples though. Take the code I have below and execute it in your console, if you want to follow along with my examples.

```
matrix1 <- matrix(c(2,0,1,3),nrow=2,ncol=2)
matrix1
matrix2 <- matrix(c(5,7),nrow=2)
matrix2
matrix6 <- matrix(c(4,3,1,3),nrow=2,ncol=2)
matrix6
```

It's important to remember the dimensions of your matrices. Attempting to use addition on two matrices without the same dimensions won't work.

Matrix 1 and 2 do not have the same dimensions, so it will return an error:

```
matrix1 + matrix2
```

```
## Error in matrix1 + matrix2: non-conformable arrays
```

However, Matrix 1 and Matrix 6 do have the same dimensions and will execute:

```
matrix1 + matrix6
```

```
##      [,1] [,2]
## [1,]    6    2
## [2,]    3    6
```

Multiplying two matrices together can be misleading. For example, using the simple * operator will merely multiply the corresponding values in two matrices with the same dimensions. Confused? Look at the two matrices below and then look at the output:

```
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

```
matrix6
```

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

```
matrix1 * matrix6
```

```
##      [,1] [,2]
## [1,]    8    1
## [2,]    0    9
```

Entry [1,1] of the first matrix is 2. Entry [1,1] of the second matrix is 4. 2 x 4 = 8. That shows us that the multiplication used here is not true matrix multiplication.

If you attempt to use the same * operator for Matrix 1 and Matrix 2 though, you will get an error:

```
matrix1
```

```
##      [,1] [,2]
## [1,]    2    1
## [2,]    0    3
```

```
matrix2
```

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