R Introduction

Will Gervais

October 2022

Contents

1 Installing R **2**

2 Installing R Studio **3**

3 R Studio Interface **4**

4 Base R and Packages **6**
4.1 Using R 7

5 Loading Data **8**
5.1 What if your data aren't in a .csv? 8
5.2 An example 8

6 Basic Data Wrangling **11**
6.1 Standardizing a Variable 11
6.2 Creating a New Variable From Existing Variables 11
6.3 Creating a New Data Frame By Subsetting An Existing Frame 12

7 Very Basic Plotting **14**
7.1 Plotting distributions for a variable 14
7.2 Plotting bivariate relations 16
7.3 ggplot: The Grammar of Graphics 16

8 Very Basic Stats **19**
8.1 Descriptives 19
8.2 Comparing means (t-tests) 19
8.3 Quick Regression 20

9 Closing **25**

This week, we'll be getting familiar with the R programming language. R can be SUPER confusing, but it's also SUPER useful. I'll be introducing it with real data from a project I ran a couple of years ago.

We'll be covering a few different things:

1) Installing R
2) Installing R Studio
3) The R/R Studio Interface
4) Base R and Packages
5) Loading Data
6) Wrangling Data (very quick 101 version)
7) Some VERY basic plots
8) (very) Basic Statistics in R

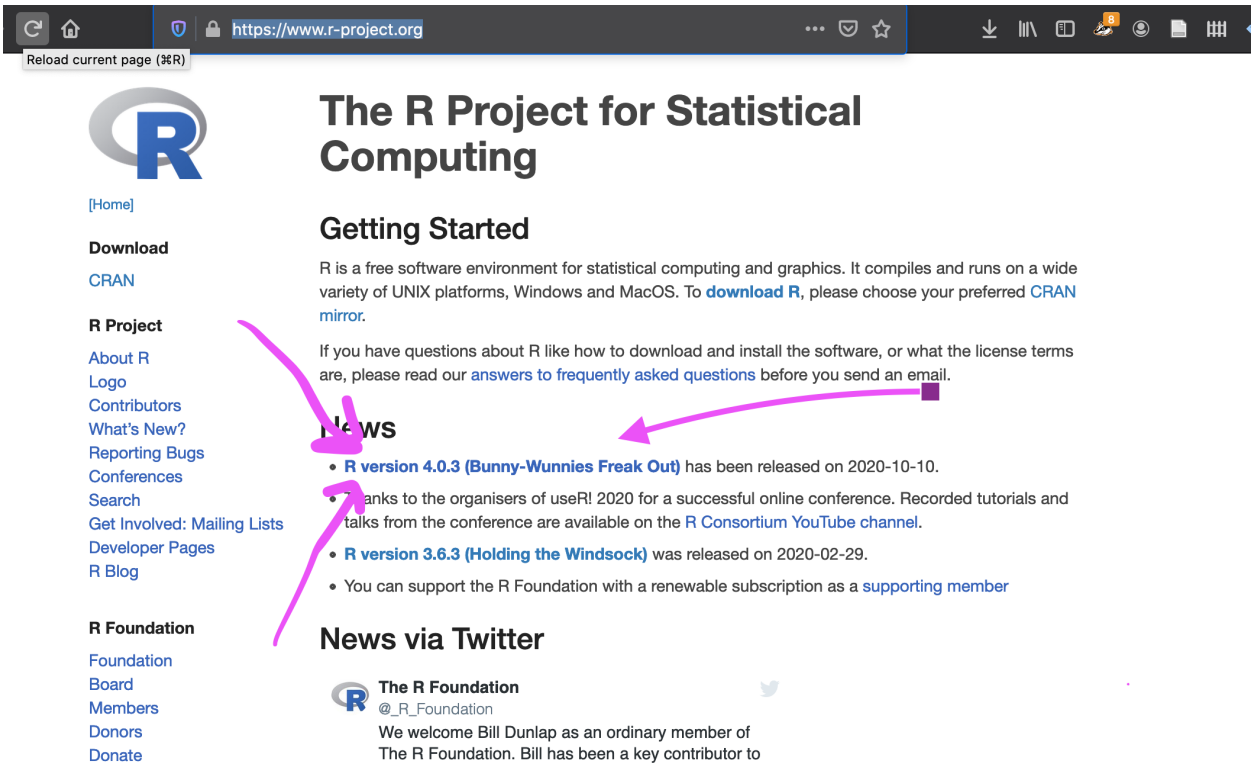
Through all of this, there are a few resources that'll be *very* helpful.

- Danielle Navarro's Learning Statistics With R book is a great resource, for all things stats and R.
- Google. No joke, I've been using R to analyze data for more than a decade and nary an analysis session goes by without me just Googling things, there's a lot of good sources out there.
- package documentation (typing `?blah` to get answer about `blah` from documentation)

1 Installing R

R is a free, open-source program. Pretty nifty!

To install it, head on over to R's Website and scroll around until you find the installation that fits your machine. Click the clicks, install it much as you would any other program.



VOILA!, you've installed R. At this point, you could just run R as is. But the base R is, IMHO, clunky and pretty terrible.

Thankfully, there's R Studio!

2 Installing R Studio

The R program is all the fancy stuff that does the statistics. But it's messy and not terribly easy to use. And the user interface is a mess. So I HIGHLY RECOMMEND you not actually use the R console for using R.

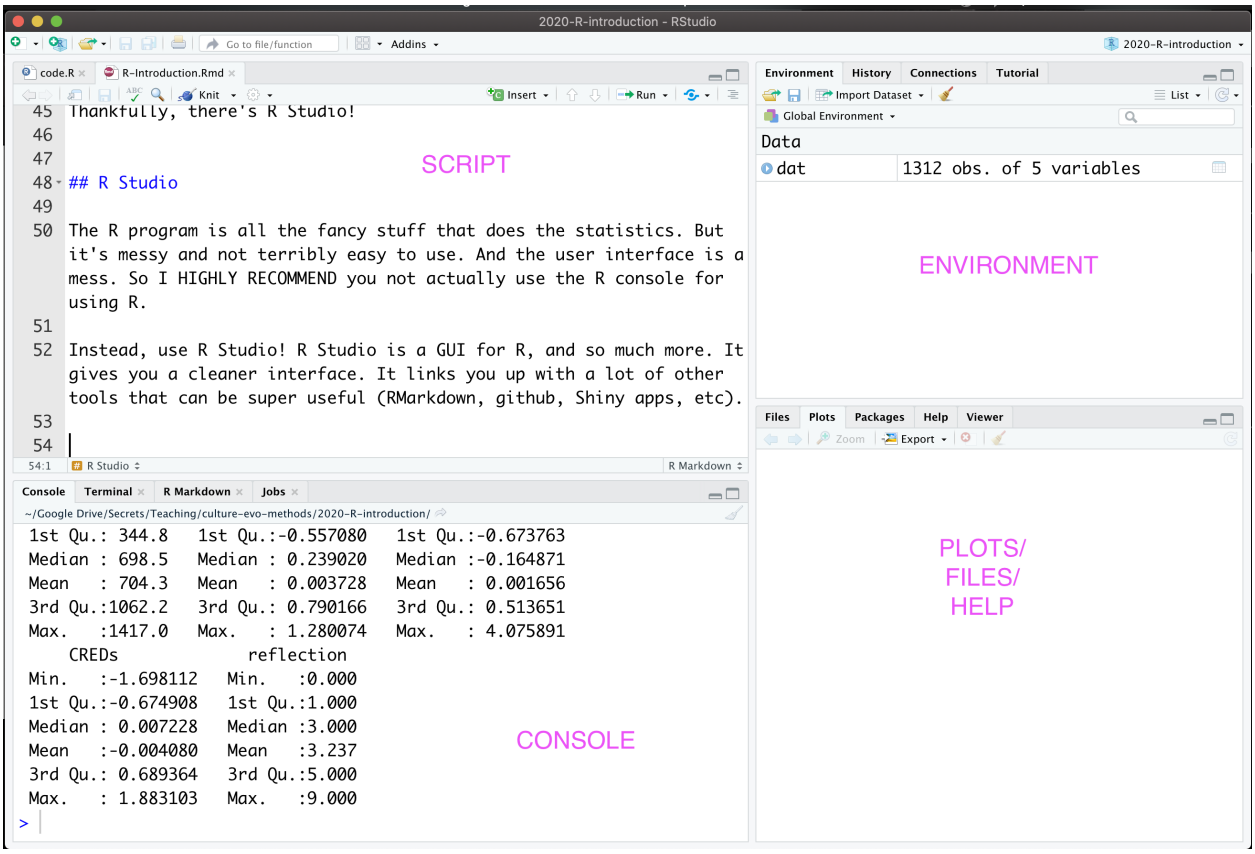
Instead, use R Studio! R Studio is a GUI for R, and so much more. It gives you a cleaner interface. It links you up with a lot of other tools that can be super useful (RMarkdown, github, Shiny apps, etc).

To install R Studio, head to R Studio's Website, navigate around to find the version for your computer. Then do your normal installation thing. Nothing fancy to sort out.

Now you have R, a very useful program. And you have R Studio, a program that makes R more tolerable to actually use. You're basically ready to go!

3 R Studio Interface

R Studio usually looks something like this:



It divides your screen into a few sections whose uses will be confusing right now, but will make sense as you use them.

- SCRIPT: This part of the screen is where you can work on and edit and run your scripts (think SPSS syntax)
- CONSOLE: your results show up here. You can also type in commands and do things here, but it won't be recorded (think: doing things in SPSS without saving to syntax)
- PLOT/etc: bottom right is where plots will show up. Plus you can check files in your directory, look for help, install packages, etc.
- ENVIRONMENT: this just shows you what sorts of things you're working with. Again, it'll make sense eventually.

Most of your R Studio life will be typing code in the Script, running it, checking the results in the Console area, and viewing plots in the bottom right.

Keep an eye on how things change as you load and play with data. These changes will be reported in the CONSOLE and they'll be reflected up in the ENVIRONMENT area. But don't stress too much about it now.

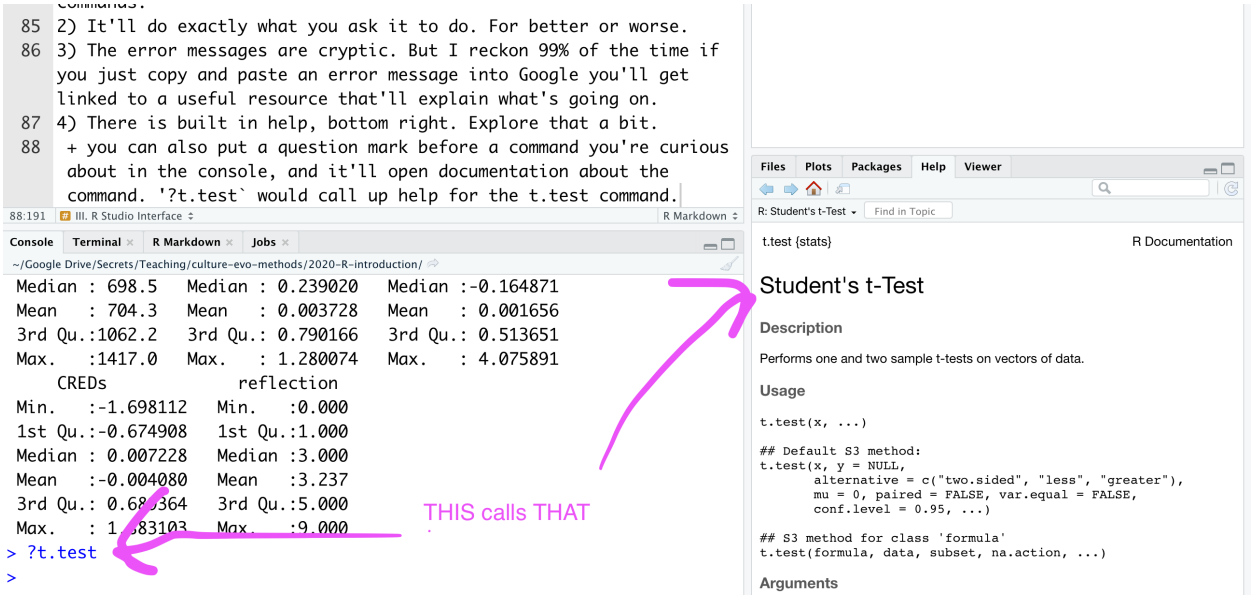
MY philosophy to learning R (and most things) is that you've just gotta dive in and try to do it. There'll be messes along the way – hopefully informative ones – but you'll quickly adapt and figure out what works and what doesn't.

A couple of important notes about coding in R/RStudio:

1) R is CASE SENSITIVE. It treats `t.test` and `T.Test` and `t.Test` as different commands.
2) It'll do exactly what you ask it to do. For better or worse.

3) The error messages are cryptic. But I reckon 99% of the time if you just copy and paste an error message into Google you'll get linked to a useful resource that'll explain what's going on.
4) There is built in help, bottom right. Explore that a bit.

- you can also put a question mark before a command you're curious about in the console, and it'll open documentation about the command. '?t.test' would call up help for the t.test command.



Poke around. Have fun. Get lost. Type some stuff in the console to see what happens. Make a mess.

4 Base R and Packages

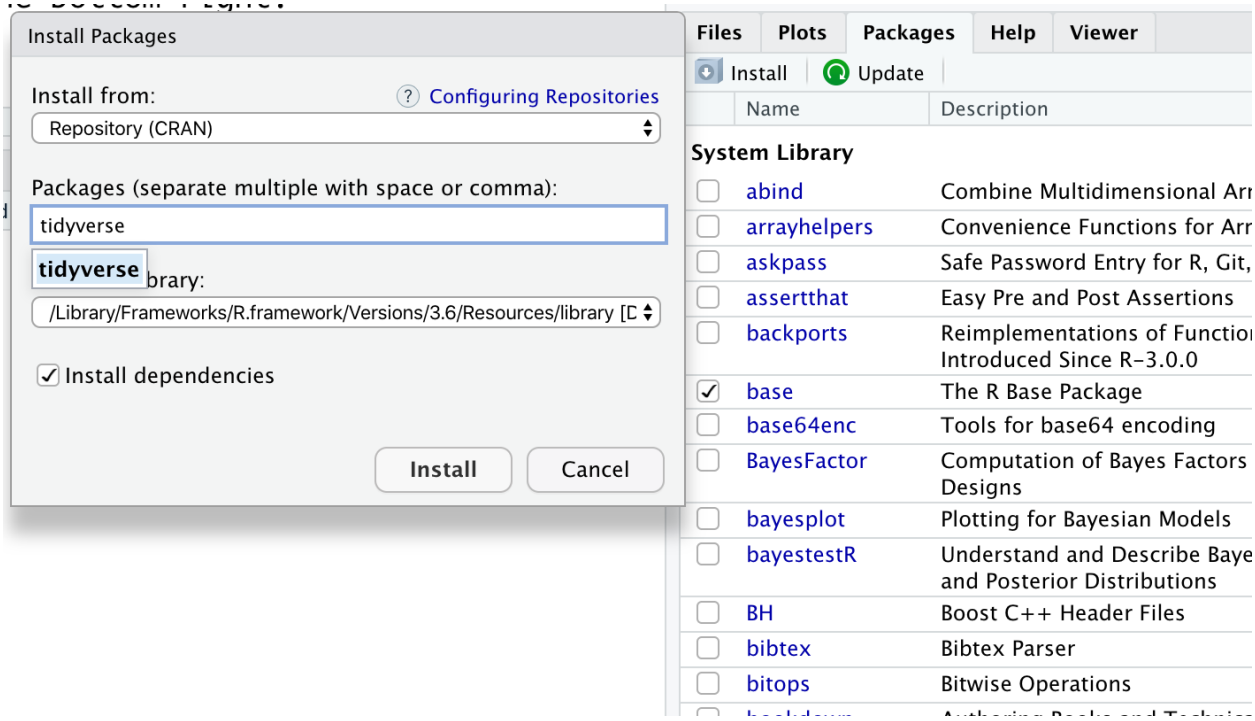
R is a programming language. But it's also the program you just downloaded. The program itself does a lot of very handy stats and plotting stuff. It does this with a bunch of built in functions. The base-level built-in stuff is called 'base R.'

But R has become a lot more than just base R over the years. Intrepid statisticians and programmers have built an increasingly complex number of add-on functions and things for specific tasks. These range from functions that can do very specific things (e.g., import data from SPSS to R) to other sets of functions that reflect whole sweeping suites of things you can do (e.g., a whole mini-language with a grammar for wrangling data [the tidyverse], or for making very nice plots [ggplot]). These add-on features to R are called **packages.** You'll end up using a lot of different packages for different sorts of data science tasks. Packages are the superpowers you can acquire in R.ss

For now, we'll keep it fairly sparse and only load up one package that has a bunch of functions for working with data and doing some plotting. It's called `tidyverse`. But `tidyverse` itself is a package clumping together a bunch of smaller packages that do their own specific things. There's one for data manipulation (`dplyr`), one for plotting (`ggplot2`), etc. It's really useful!

There are two ways to load packages.

One is with the drop downs at the bottom right.



The other option is to do it with code, either in the console or script.

To do that, you use the following code:

```
install.packages("tidyverse")

library(tidyverse)
```

That first command (`install.packages()`) does the actual installing. But it's not enough to just install the package. . . you'll also have to call the package when you need it, usually once per script. That's what the second command there (`library()`) is doing.

I use tidyverse in like 99% of my analyses, so the top line of my scripts usually call `tidyverse` with the `library()` function. As you use R more, you'll figure out which packages you use a ton and which you basically never need. So you can tailor each analysis script according to what you'll need.

4.1 Using R

R has a bewildering number of functions and features. Toss in some packages and it's even more bewildering. But at its core, it's just a programming langauge. There are some nifty features to this language. We won't have time to cover all the coding nuts and bolts, but Navarro's book has an excellent introduction. Chapters 3 and 4 are the programming crash course you need.

In the meantime, there's a couple of things you'll find yourself doing A TON in R. The main one is *assigning*

At its core, *assigning* is just giving things names that you'll work with in R. You can assign variables, whole data frames, or single values.

For example, this code simply "assigns" the value 5 to x:

```
x <- 5
```

If you've done that, then it will forever treat 'x' as '5'

Try it!

```
x <- 5

y <- 7

x + y
```

```
## [1] 12
```

Here we've told R that 'x' means 5 now, and 'y' means 7 now. So when we ask for x + y it tells us it's 12.

This is trivially simple in this case, but the assign operator `<-` is basically just telling R that A THING on the right now goes by the name on the left. So it could be a whole column in a dataset, or a whole dataset. That sounds cryptic now but it'll make sense in a minute. I promise.

We just need some data to make this concrete.

5 Loading Data

Full stop, the easiest way to use and store data for use with R is with .csv files. If you're pulling data from Qualtrics or wherever, just pop it in Excel (or comparable) and save it as a .csv. Then it's a cinch to call the data into R. You basically just have to tell R where the file is and it'll load right away. So set your working directory to wherever your data are located. You can do this with a dropdown menu under "Session". Or you can do it with the `setwd()` function.

```
setwd("~/Wherever/You/Put/The/File/")
```

Then you simply ask it to call up the file you want and put it in a data frame in R. Data frames are the bread and butter way to deal with data sets. THink basically of a spreadsheet, like the .dat part of SPSS.

```
data <- read.csv("your-data.csv")
```

This code would read in "your-data.csv" to R as a data frame that you've cleverly named `data`. If you do this, you'll notice that your data frame named `data` now shows up in the top right in the ENVIRONMENT. This just means you've got a thing in R called `data`.

5.1 What if your data aren't in a .csv?

If, as is often the case, your data are in some format other than a CSV, don't fret. You can still get it into R. You could do this by directly saving your data set (from SPSS, SYSTAT, STATA, etc) as a .csv then doing the above. Or, you can use one of many packages out there that make this easy. I recommend the `foreign` and `haven` packages for these tasks. Using `foreign`, here's the code to grab an SPSS file and get it into R.

```
install.packages("foreign")
library(foreign)

data <- read.spss("your-data.sav", to.data.frame=TRUE)
```

5.2 An example

For the rest of this intro, let's play with some real data. I sent out a .csv file called "religion.csv" and it's got just a few variables from a big dataset I collected a few years back. Let's get it into R.

1) Set your working directory wherever you saved the file.

```
setwd("~/Where/Did/You/Put/It")
```

2) read that CSV, assign it to a data frame called "dat" (or whatever else you want to call your data frame)

```
dat <- read.csv("religion.csv")
```

3) then, you can do some things to check it out. You can..

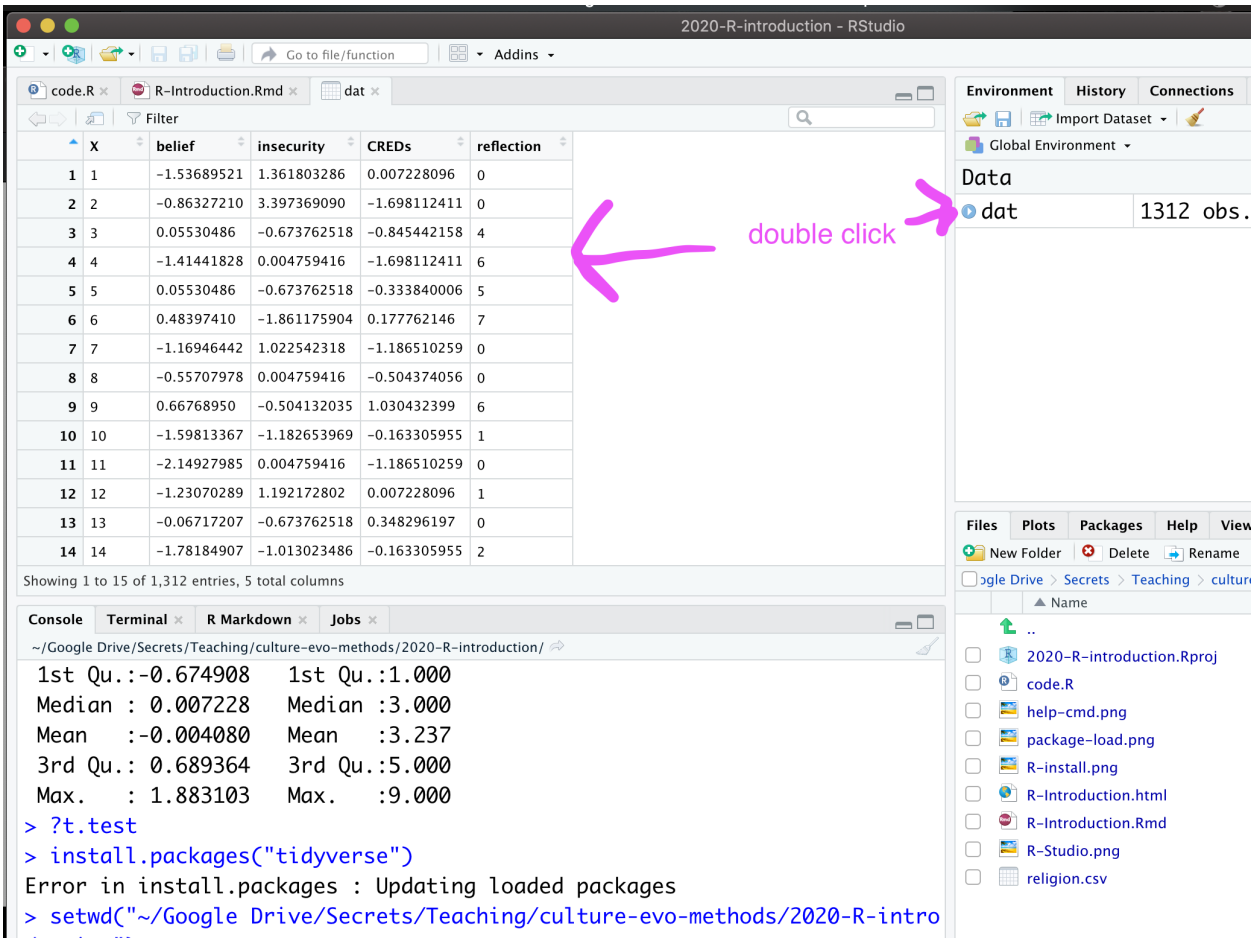
- look at the whole thing
- look at the first few rows
- get a summary

What happens if you just ask for the whole thing?

```
dat
```

Dang, what a mess. Do you really want to see a gross printout of your whole dataset? Not likely.

Instead, double click on the name of your data frame in the environment if you want to see the data frame, like you might in SPSS.



If you just want a sanity check that everything's there, you can usually get by just taking a quick glance at the first few rows. To do that, use the `head()` function.

```
head(dat)
```

```
##   X      belief   insecurity        CREDs reflection
## 1 1 -1.53689521  1.361803286  0.007228096          0
## 2 2 -0.86327210  3.397369090 -1.698112411          0
## 3 3  0.05530486 -0.673762518 -0.845442158          4
## 4 4 -1.41441828  0.004759416 -1.698112411          6
## 5 5  0.05530486 -0.673762518 -0.333840006          5
## 6 6  0.48397410 -1.861175904  0.177762146          7
```

Or if you want a quick rundown summary of the variables, try `summary()`

```
summary(dat)
```

```
##        X              belief            insecurity           CREDs
##  Min.   :   1.0   Min.   :-2.394234   Min.   :-2.370067   Min.   :-1.698112
##  1st Qu.: 344.8   1st Qu.:-0.557080   1st Qu.:-0.673763   1st Qu.:-0.674908
##  Median : 698.5   Median : 0.239020   Median :-0.164871   Median : 0.007228
##  Mean   : 704.3   Mean   : 0.003728   Mean   : 0.001656   Mean   :-0.004080
##  3rd Qu.:1062.2   3rd Qu.: 0.790166   3rd Qu.: 0.513651   3rd Qu.: 0.689364
##  Max.   :1417.0   Max.   : 1.280074   Max.   : 4.075891   Max.   : 1.883103
##    reflection
##  Min.   :0.000
```

```
##  1st Qu.:1.000
##  Median :3.000
##  Mean   :3.237
##  3rd Qu.:5.000
##  Max.   :9.000
```

You can learn a lot about your variables this way. Eyeball both the head rows and the summary, and you'll see that the "belief", "insecurity", "CREDs" variables are all continuous. Based on values and mean ~0, you might guess (correctly!) that these were continuous variables that have been standardized. "reflection" on the other hand looks like a count variable.

At this point, you've basically gotten to the point of having a data set in R. Now let's do stuff with it!

6 Basic Data Wrangling

I'd estimate that more than half of the time I spend with data is just wrangling it and massaging it and cleaning it up to get it to the point where you can analyze it. The set I gave y'all is already fairly tidy. I've compiled index variables, cleared up missing values, etc. But let's play a bit to see what we can do.

Base R has a lot of functionality for wrangling, but I think the `tidyverse` suite really comes into its own here. So I'll use tidyverse code here. But there are base R analogues for all of this.

6.1 Standardizing a Variable

Oftentimes you want to transform a variable in one way or another. For example, you might want to standardize it (z-score it). There's a nice function that can do this for you called `scale`.

Let's standardize the CREDs count variable so it's in the same scale as the rest of the variables. To do so, we are going to take our data frame, tell R that we want to generate a new variable "reflectionZ", and that we want to do so by rescaling an existing variable "reflection"...

```
dat <- dat %>%
  mutate(reflectionZ = scale(reflection, scale=T, center=T)[,])
```

Let's check to see if it worked, via `summary` or `head`

```
head(dat)
```

```
##   X      belief   insecurity        CREDs reflection reflectionZ
## 1 1 -1.53689521  1.361803286  0.007228096          0  -1.2058586
## 2 2 -0.86327210  3.397369090 -1.698112411          0  -1.2058586
## 3 3  0.05530486 -0.673762518 -0.845442158          4   0.2842158
## 4 4 -1.41441828  0.004759416 -1.698112411          6   1.0292530
## 5 5  0.05530486 -0.673762518 -0.333840006          5   0.6567344
## 6 6  0.48397410 -1.861175904  0.177762146          7   1.4017716
```

```
summary(dat)
```

```
##        X              belief            insecurity            CREDs
##  Min.   :   1.0   Min.   :-2.394234   Min.   :-2.370067   Min.   :-1.698112
##  1st Qu.: 344.8   1st Qu.:-0.557080   1st Qu.:-0.673763   1st Qu.:-0.674908
##  Median : 698.5   Median : 0.239020   Median :-0.164871   Median : 0.007228
##  Mean   : 704.3   Mean   : 0.003728   Mean   : 0.001656   Mean   :-0.004080
##  3rd Qu.:1062.2   3rd Qu.: 0.790166   3rd Qu.: 0.513651   3rd Qu.: 0.689364
##  Max.   :1417.0   Max.   : 1.280074   Max.   : 4.075891   Max.   : 1.883103
##    reflection     reflectionZ
##  Min.   :0.000   Min.   :-1.2059
##  1st Qu.:1.000   1st Qu.:-0.8333
##  Median :3.000   Median :-0.0883
##  Mean   :3.237   Mean   : 0.0000
##  3rd Qu.:5.000   3rd Qu.: 0.6567
##  Max.   :9.000   Max.   : 2.1468
```

Cool! We've got a new variable!

6.2 Creating a New Variable From Existing Variables

Let's say we want to create a new variable that's the average score from, say, reflectionZ and CREDs and insecurity. That would look like this:

```
dat <- dat %>%
  mutate(avg = (insecurity + CREDs + reflectionZ)/3)
```

```
head(dat)
```

```
##   X      belief   insecurity        CREDs reflection reflectionZ         avg
## 1 1 -1.53689521  1.361803286  0.007228096          0  -1.2058586  0.05439093
## 2 2 -0.86327210  3.397369090 -1.698112411          0  -1.2058586  0.16446603
## 3 3  0.05530486 -0.673762518 -0.845442158          4   0.2842158 -0.41166296
## 4 4 -1.41441828  0.004759416 -1.698112411          6   1.0292530 -0.22136667
## 5 5  0.05530486 -0.673762518 -0.333840006          5   0.6567344 -0.11695605
## 6 6  0.48397410 -1.861175904  0.177762146          7   1.4017716 -0.09388073
```

`mutate` lets you nest a bunch of these transmformations together. Here's some nonsensical code to make a variable that's the square root of "reflection", another that multiplies that by "CREDs", and one that mean-centers (but doesn't standardize) that...

```
dat <- dat %>%
  mutate(refSR = sqrt(reflection),
         rXCREDs = refSR * CREDs,
         rXCREDsZero = scale(rXCREDs, center=T, scale=F)[,])

summary(dat)
```

```
##        X              belief            insecurity            CREDs
##  Min.   :   1.0   Min.   :-2.394234   Min.   :-2.370067   Min.   :-1.698112
##  1st Qu.: 344.8   1st Qu.:-0.557080   1st Qu.:-0.673763   1st Qu.:-0.674908
##  Median : 698.5   Median : 0.239020   Median :-0.164871   Median : 0.007228
##  Mean   : 704.3   Mean   : 0.003728   Mean   : 0.001656   Mean   :-0.004080
##  3rd Qu.:1062.2   3rd Qu.: 0.790166   3rd Qu.: 0.513651   3rd Qu.: 0.689364
##  Max.   :1417.0   Max.   : 1.280074   Max.   : 4.075891   Max.   : 1.883103
##    reflection     reflectionZ           avg                 refSR
##  Min.   :0.000   Min.   :-1.2059   Min.   :-1.5207529   Min.   :0.000
##  1st Qu.:1.000   1st Qu.:-0.8333   1st Qu.:-0.3755594   1st Qu.:1.000
##  Median :3.000   Median :-0.0883   Median :-0.0243843   Median :1.732
##  Mean   :3.237   Mean   : 0.0000   Mean   :-0.0008079   Mean   :1.560
##  3rd Qu.:5.000   3rd Qu.: 0.6567   3rd Qu.: 0.3586705   3rd Qu.:2.236
##  Max.   :9.000   Max.   : 2.1468   Max.   : 1.5843784   Max.   :3.000
##     rXCREDs          rXCREDsZero
##  Min.   :-5.09434   Min.   :-5.16783
##  1st Qu.:-0.87360   1st Qu.:-0.94709
##  Median : 0.00000   Median :-0.07349
##  Mean   : 0.07349   Mean   : 0.00000
##  3rd Qu.: 1.03043   3rd Qu.: 0.95694
##  Max.   : 5.64931   Max.   : 5.57582
```

For `mutate` really it'll do any transformation you can think of and code. It's super flexible.

6.3 Creating a New Data Frame By Subsetting An Existing Frame

Sometimes you want to run analyses on only a portion of the data. Maybe you've got a filtering variable for people who passed an attention check. Maybe you need to analyze different nationalities separately. Maybe you need to toss out really old (or young) people.

That's easy to do with `filter`. Here we can filter to only **include** rows where people scored at least 2 on the 'reflection' variable. Check out the summary of the resulting frame `dat2`, and `nrow` also lets you check how many rows are in each of your data frames. The filtering, as you can see, dropped a whole bunch of cases.

```
dat2 <- dat %>%
  filter(reflection >= 2)

summary(dat2)
```

```
##        X              belief          insecurity          CREDs
##  Min.   :   3.0   Min.   :-2.3942   Min.   :-2.3701   Min.   :-1.698112
##  1st Qu.: 338.0   1st Qu.:-0.5571   1st Qu.:-0.6738   1st Qu.:-0.674908
##  Median : 675.0   Median : 0.2390   Median :-0.1649   Median : 0.007228
##  Mean   : 690.1   Mean   :-0.0545   Mean   :-0.1478   Mean   : 0.058291
##  3rd Qu.:1037.5   3rd Qu.: 0.7902   3rd Qu.: 0.3440   3rd Qu.: 0.859898
##  Max.   :1417.0   Max.   : 1.2801   Max.   : 3.0581   Max.   : 1.883103
##    reflection     reflectionZ           avg              refSR
##  Min.   :2.000   Min.   :-0.4608   Min.   :-1.1704   Min.   :1.414
##  1st Qu.:3.000   1st Qu.:-0.0883   1st Qu.:-0.1961   1st Qu.:1.732
##  Median :4.000   Median : 0.2842   Median : 0.1218   Median :2.000
##  Mean   :4.592   Mean   : 0.5047   Mean   : 0.1384   Mean   :2.077
##  3rd Qu.:6.000   3rd Qu.: 1.0293   3rd Qu.: 0.4823   3rd Qu.:2.449
##  Max.   :9.000   Max.   : 2.1468   Max.   : 1.4693   Max.   :3.000
##     rXCREDs          rXCREDsZero
##  Min.   :-5.09434   Min.   :-5.16783
##  1st Qu.:-1.23546   1st Qu.:-1.30895
##  Median : 0.02044   Median :-0.05305
##  Mean   : 0.12565   Mean   : 0.05216
##  3rd Qu.: 1.68859   3rd Qu.: 1.61510
##  Max.   : 5.64931   Max.   : 5.57582
```

```
nrow(dat)
```

```
## [1] 1312
```

```
nrow(dat2)
```

```
## [1] 875
```

You can also create a new frame that selects only a subset of columns. Oftentimes this is handy when you're working with a gigantic dataset with lots of variables you don't really care about. Let's say you just want belief, insecurity, CREDs, and our standardized reflection variable.

```
dat3 <- dat %>%
  select(belief,
         insecurity,
         CREDs,
         ref = reflectionZ) # the = renames the variable as we're subsetting

head(dat3)
```

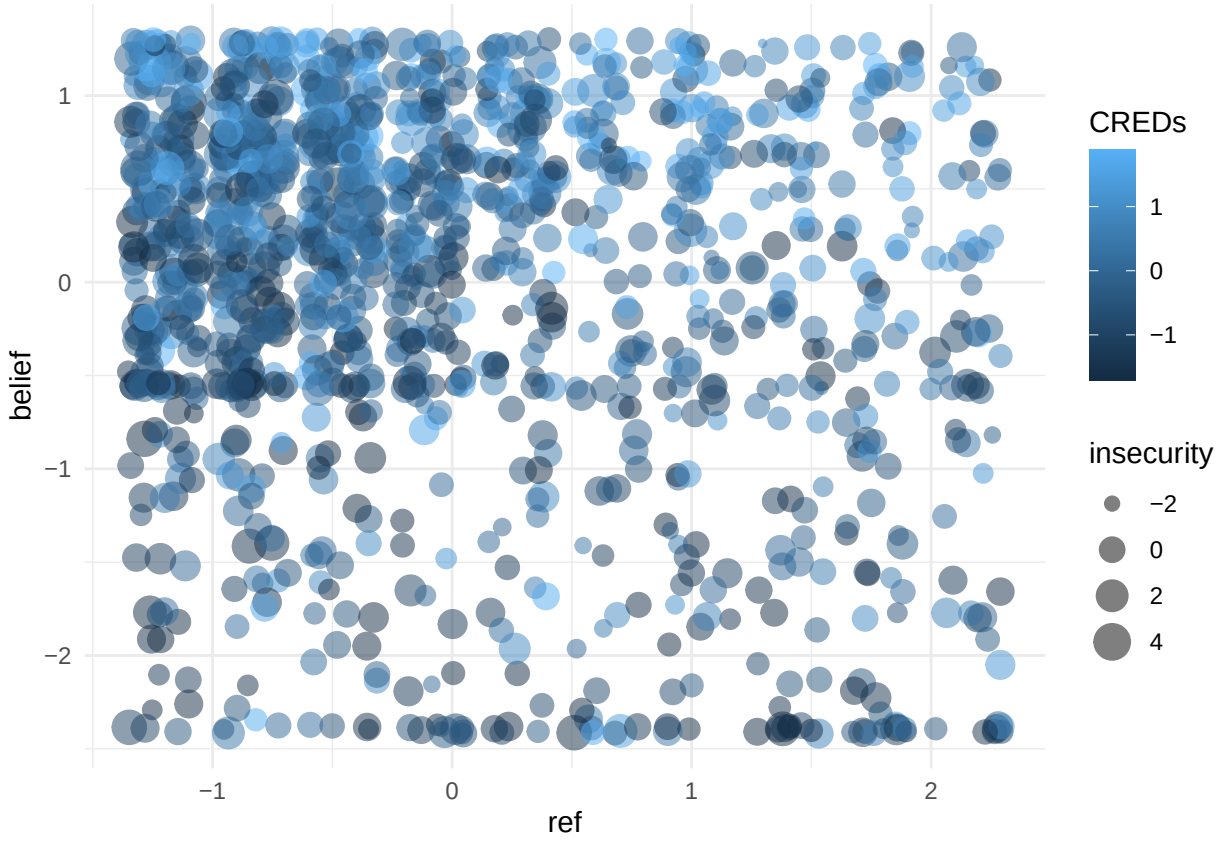
```
##        belief   insecurity        CREDs        ref
## 1 -1.53689521  1.361803286  0.007228096 -1.2058586
## 2 -0.86327210  3.397369090 -1.698112411 -1.2058586
## 3  0.05530486 -0.673762518 -0.845442158  0.2842158
## 4 -1.41441828  0.004759416 -1.698112411  1.0292530
## 5  0.05530486 -0.673762518 -0.333840006  0.6567344
## 6  0.48397410 -1.861175904  0.177762146  1.4017716
```

We'll use the `dat3` frame for some very basic plotting and stats.

7 Very Basic Plotting

A real strength of R is its data visualization capabilities. For example, here's a scatterplot of belief against reflection, colored according to CREDs, points sized according to insecurity:

```
ggplot(dat3, aes(x = ref, y = belief, size = insecurity, color = CREDs)) +
  geom_point(alpha = .5, position="jitter") +
  theme_minimal()
```

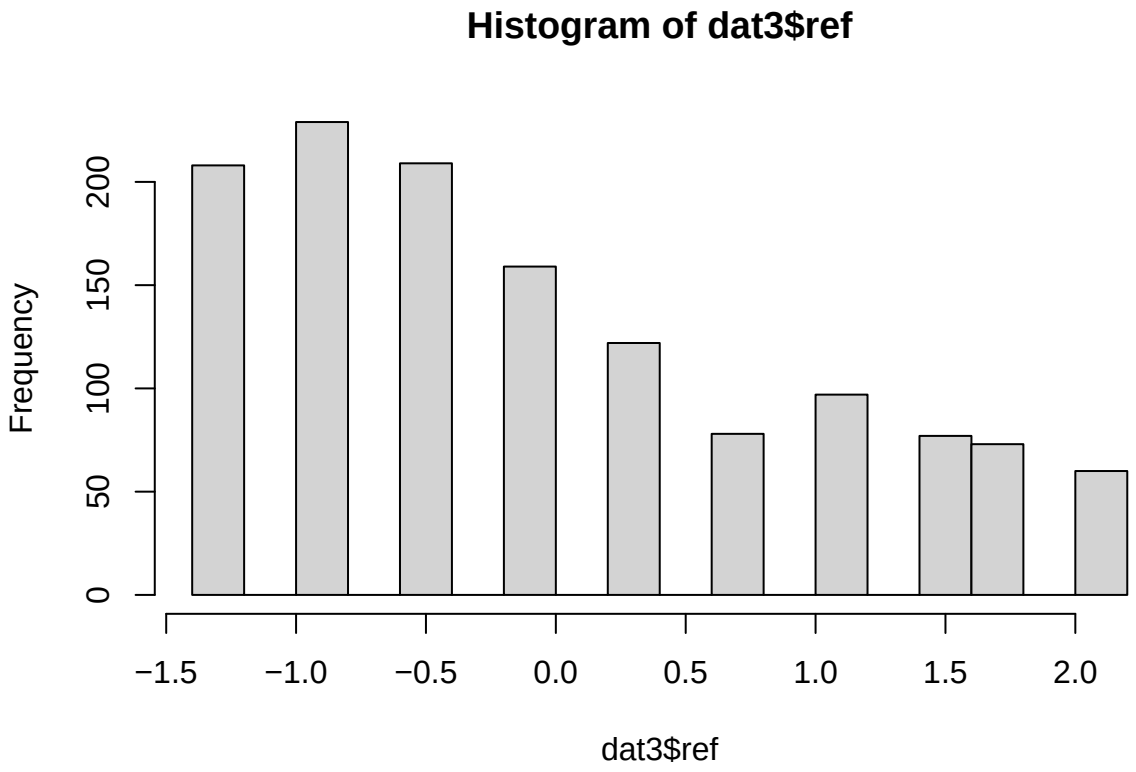


We could do a whole module on data visualization in R. For now, here are some really basic things you might want to do to just eyeball data.

7.1 Plotting distributions for a variable

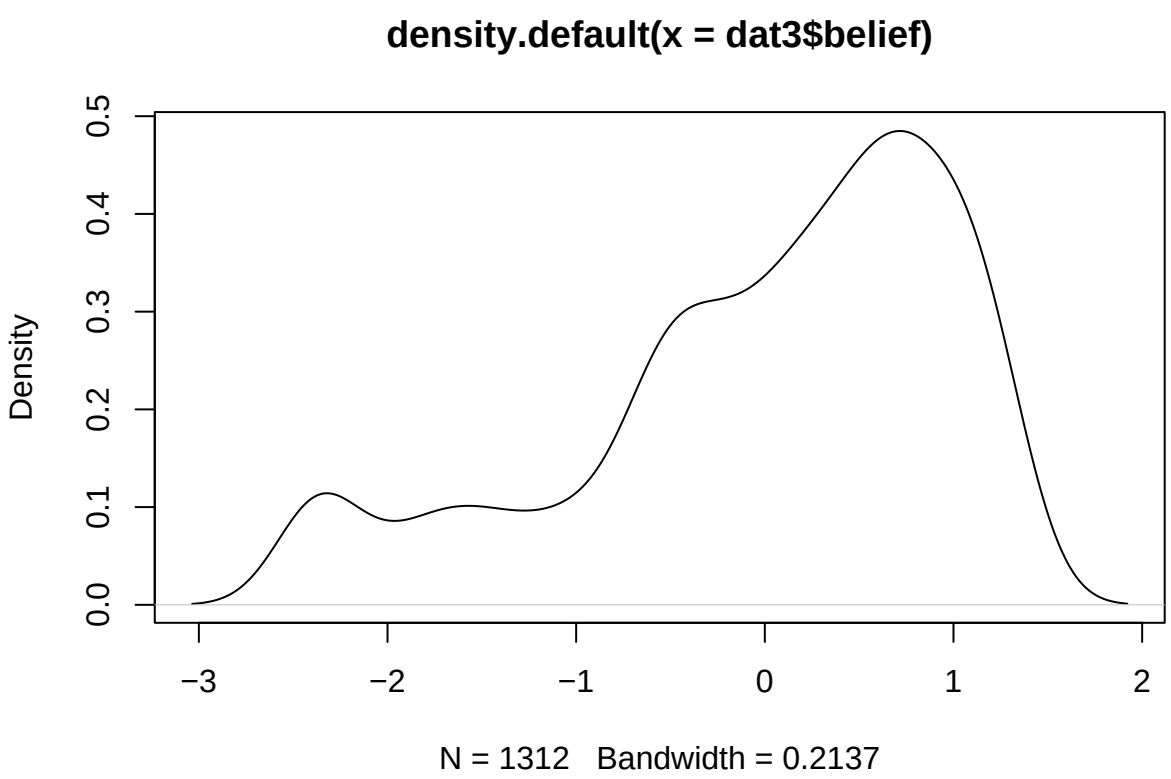
There are different ways to do this, but for quickest and dirtiest, here's the easiest:

```
hist(dat3$ref) # note: dat3$ref means "ref variable from dat3 frame"
```



You could do a density plot (think smooth histogram) like so:

```
plot(density(dat3$belief))
```

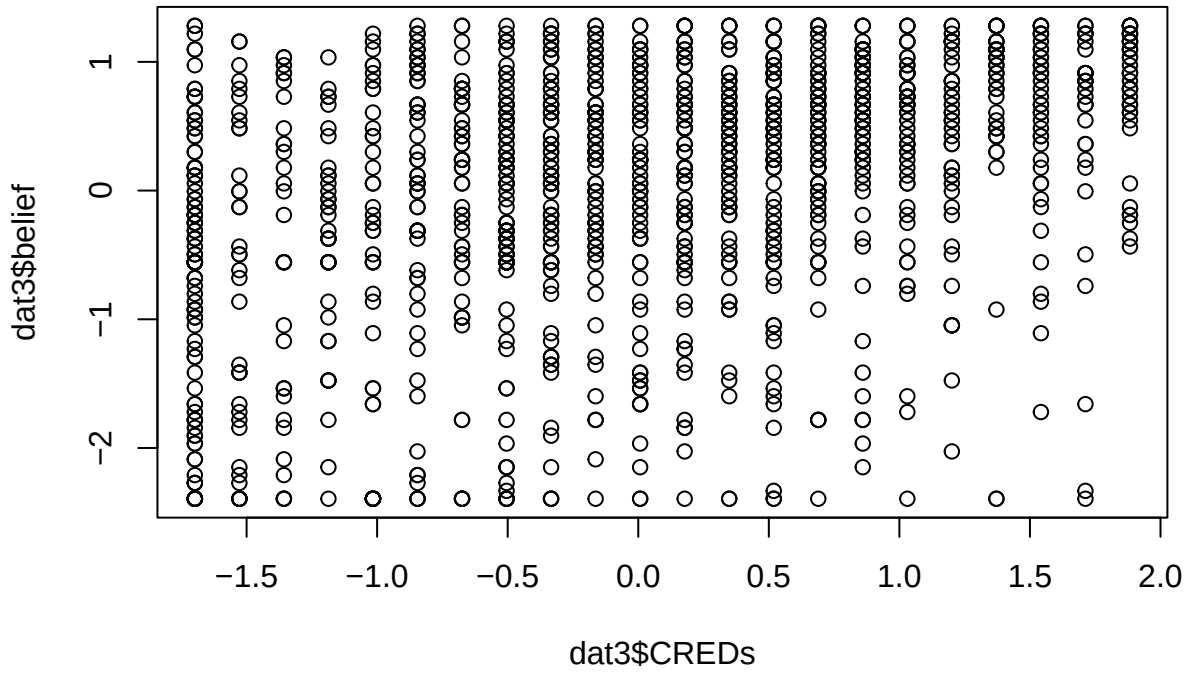


7.2 Plotting bivariate relations

For a lot of correlational work, you're interested in relations between two variables. So some form of scatterplot is nice.

The base R version of a scatterplot is pretty easy, you just have to specify which variables.

```
plot(dat3$CREDs, dat3$belief)
```

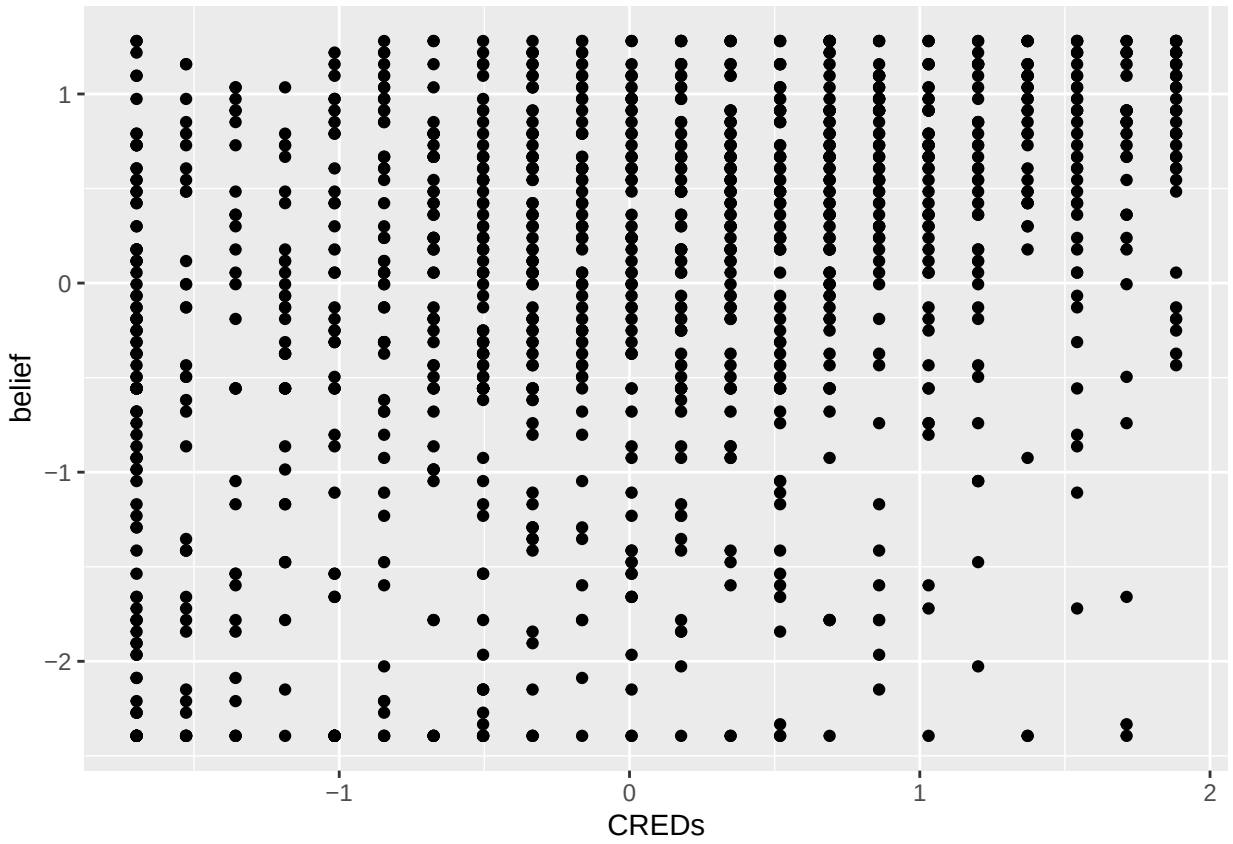


7.3 ggplot: The Grammar of Graphics

Aside from base R plotting, there's also the ggplot package (included in `tidyverse`) for building your plots up according to their "grammar of graphics" principles. Basically it means prettier graphs, way more customization, but also potentially way more code.

Here's the same scatterplot.

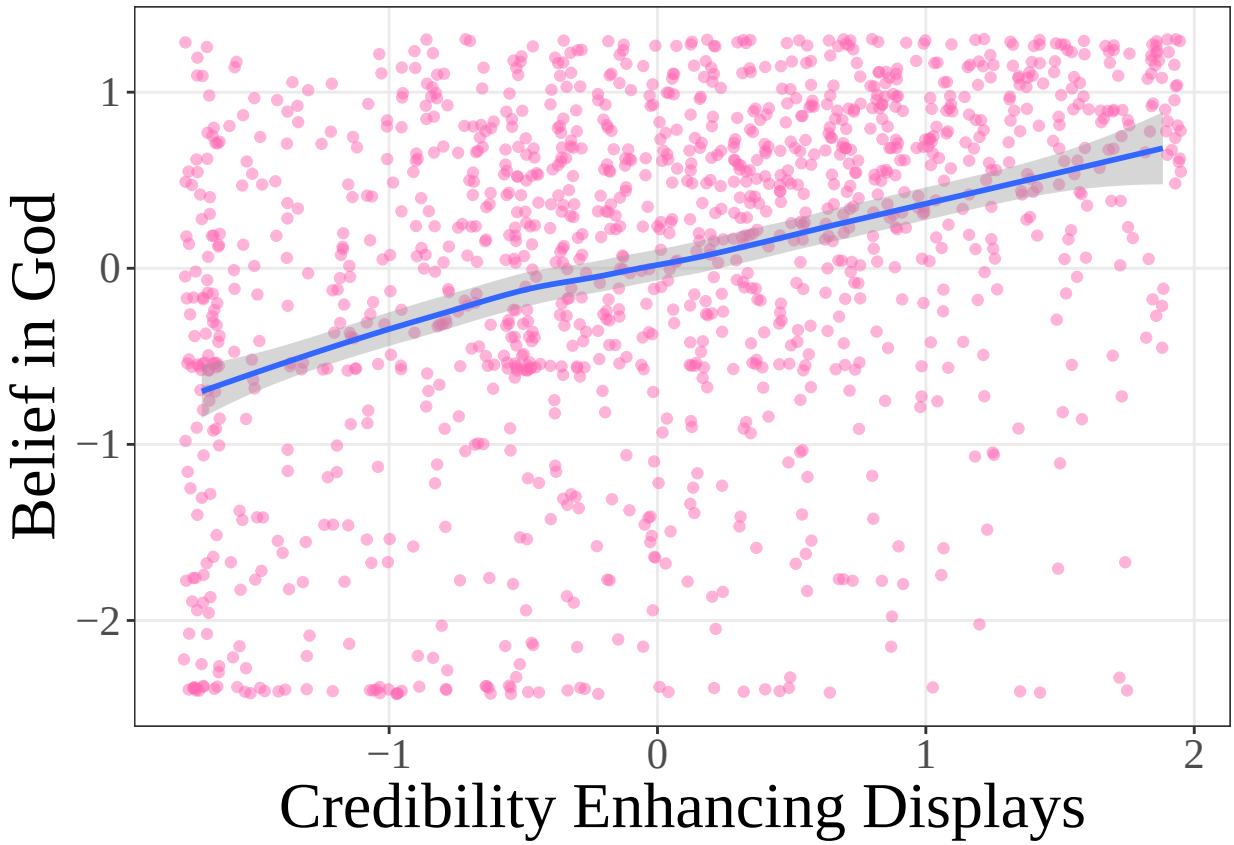
```
ggplot(dat3, aes(x=CREDs, y = belief)) +  # first you tell it what goes where
  geom_point()  # then you tell it what to draw. in this case points
```



Like I said, if you want it prettier, prepare for a lot more code.

```
ggplot(dat3, aes(x=CREDs, y = belief)) +
  geom_point(alpha=.5, position="jitter", color = "hotpink") +
  stat_smooth(method = "loess", se = T) +
  labs(x="Credibility Enhancing Displays", y = "Belief in God") +
  theme_bw() +
  theme(text = element_text(family = "Times"),
        axis.title = element_text(size = 24),
        axis.text = element_text(size = 16),
        panel.grid.minor.x = element_blank(),
        panel.grid.minor.y = element_blank(),)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Just a teaser there. Like I said, one could do a whole course on visualization in ggplot. There are some great resources out there for using ggplot though, including R Graphics Cookbook. Check it out, and enjoy.

8 Very Basic Stats

Finally, a teaser of some pretty basic ways to do stats in R.

8.1 Descriptives

There are a zillion different ways to get descriptive statistics in R. There's a single function in the `psych` package that gives you a lot of mileage.

```
library(psych)
```

```
##
## Attaching package: 'psych'
```

```
## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

```
describe(dat3)
```

```
##            vars    n mean   sd median trimmed  mad   min  max range  skew
## belief        1 1312    0 1.00   0.24    0.12 0.91 -2.39 1.28  3.67 -0.89
## insecurity    2 1312    0 1.01  -0.16   -0.05 1.01 -2.37 4.08  6.45  0.55
## CREDs         3 1312    0 1.00   0.01    0.00 1.01 -1.70 1.88  3.58 -0.01
## ref           4 1312    0 1.00  -0.09   -0.09 1.10 -1.21 2.15  3.35  0.63
##            kurtosis   se
## belief        -0.04 0.03
## insecurity     0.38 0.03
## CREDs         -0.87 0.03
## ref           -0.73 0.03
```

You can also ask for specific things, like say the median of CREDs:

```
median(dat3$CREDs)
```

```
## [1] 0.007228096
```

. . . or the standard deviation of belief:

```
sd(dat3$belief)
```

```
## [1] 0.9980419
```

8.2 Comparing means (t-tests)

If you want to compare group means, R has a built-in t-test that does what you want. For our purposes, let's create a grouping variable by arbitrarily splitting 'reflection' and then do a t-test in belief based on that. Note: this would be a statistical abomination, as far as data analysis goes. . . I'm just using it to illustrate.

```
dat3 <- dat3 %>% mutate(split = ifelse(ref > 2, 1, 0)) # new variable with a split at 2 on reflection

t.test(belief ~ split, data = dat3)
```

```
##
##  Welch Two Sample t-test
##
## data:  belief by split
## t = 3.099, df = 63.294, p-value = 0.002896
## alternative hypothesis: true difference in means between group 0 and group 1 is not equal to 0
## 95 percent confidence interval:
```

```
##  0.1650693 0.7643010
## sample estimates:
## mean in group 0 mean in group 1
##       0.0249791      -0.4397061
```

Yay! There's a statistically significant difference in belief in God between our two arbitrary groups, p = .002!

A few of notes here:

1) Get used to this format for formulae in tests you're asking for. It's always DV ~ PREDICTOR(s)
2) R defaults to Welch's t-test, which doesn't assume equal variances.
3) You get a lot of output by default.

8.3 Quick Regression

The next 2 weeks are multilevel modeling. The building block for that is regression.

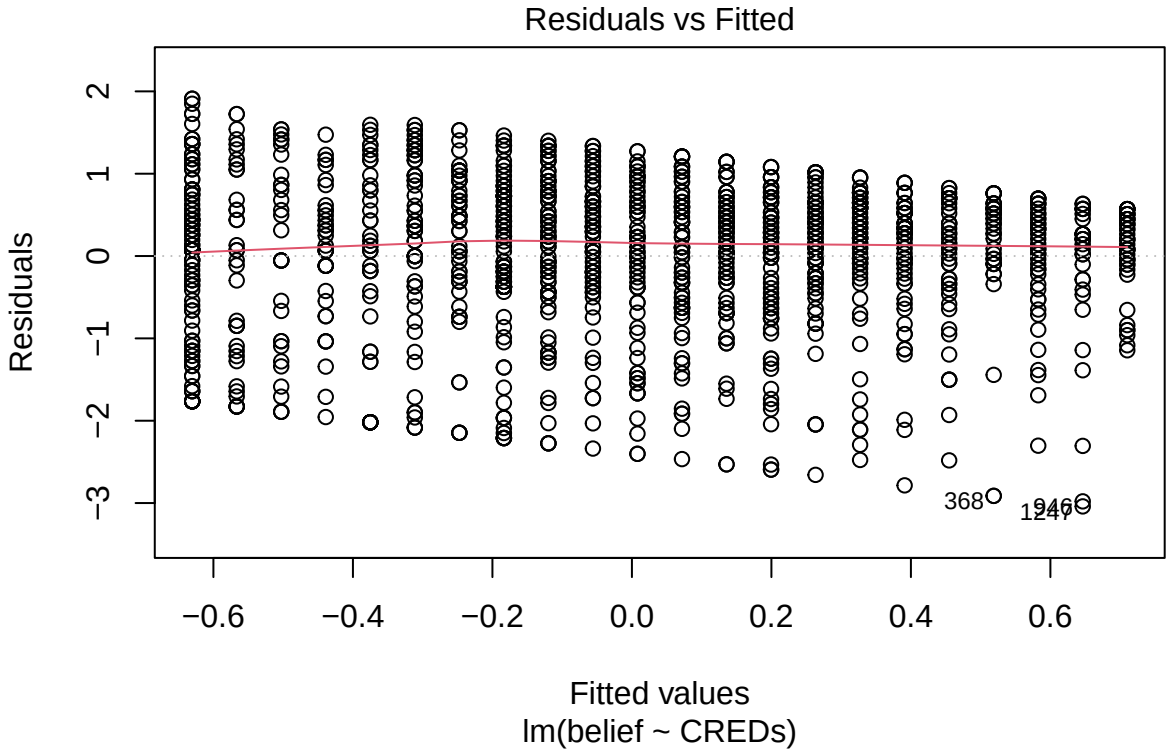
So, how do we do multiple regression in R? It's super easy. Just remember the formula setup we used for t-tests: DV ~ PREDICTOR(s).

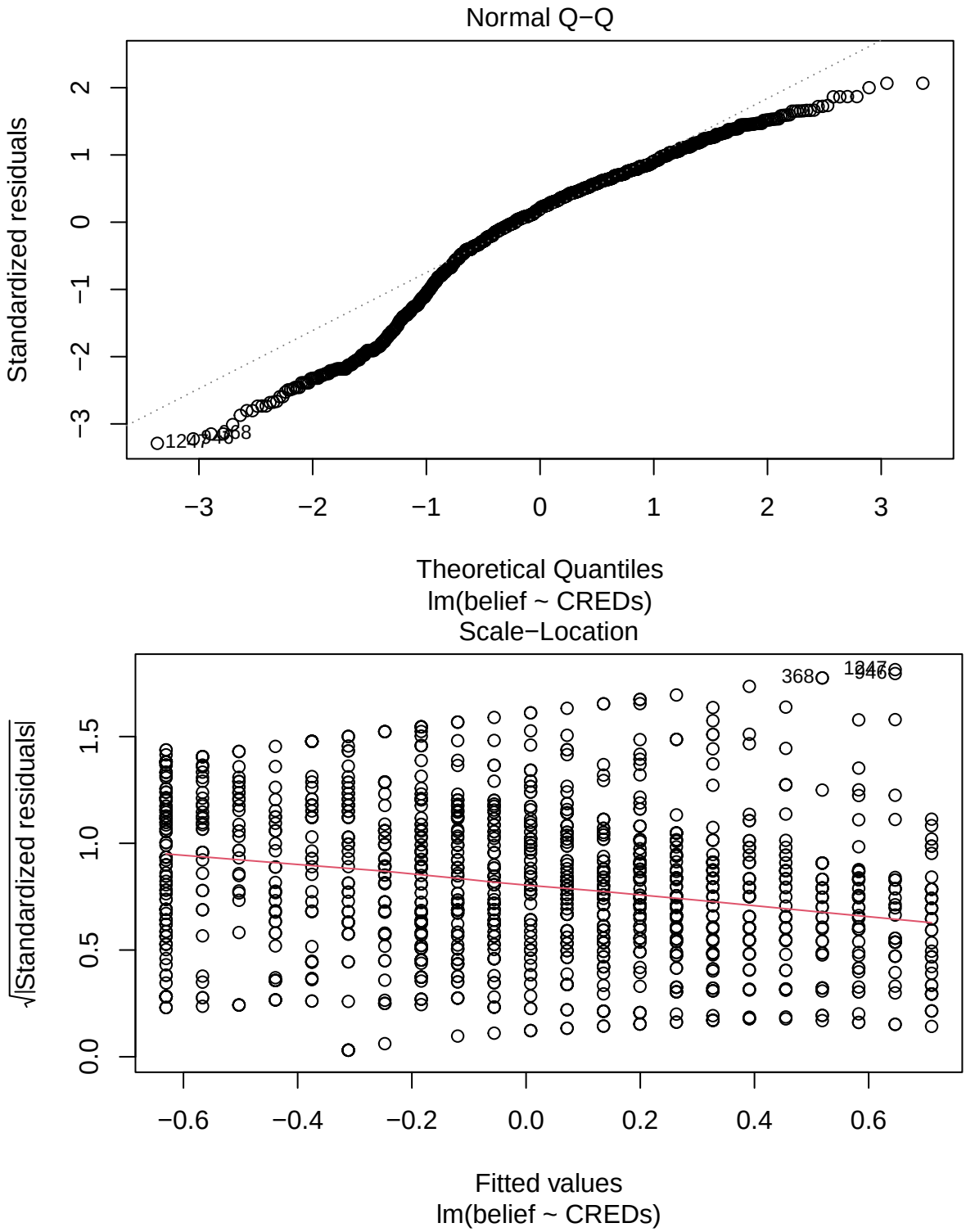
If we want, for example, to do a linear regression predicting 'belief' from 'CREDs' it would look like this: `lm(belief ~ CREDs, data = dat3)`. We want to store each model we run, for many useful reasons you'll see.

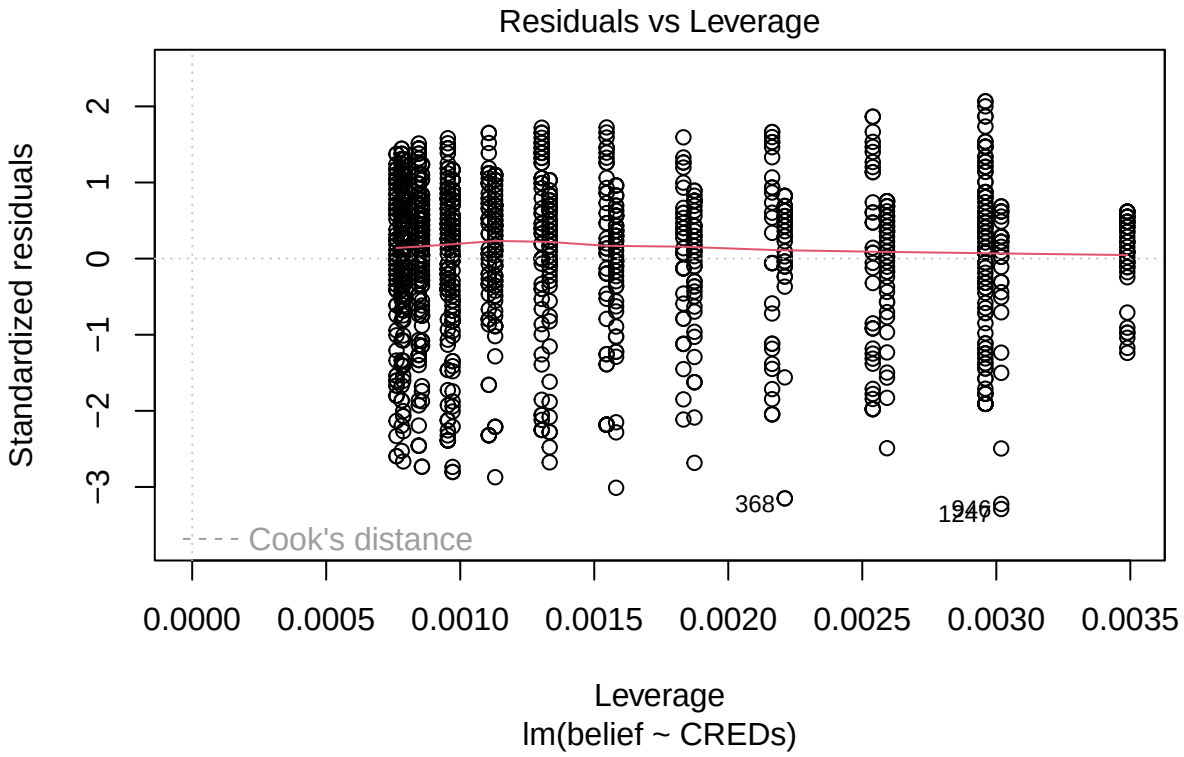
```
model1 <- lm(belief ~ CREDs, data = dat3)
```

On its own, check out what happens in the console when you do this. . . .NOTHING! You've run the model and stored all the model guts and results as the object `model1`. Now you can call various things to check out the model. You might want to check out some diagnostic plots to see if your variables were okay and the analysis made sense.

```
plot(model1)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(belief ~ CREDs)
Scale−Location
√|Standardized residuals|
Fitted values
lm(belief ~ CREDs)



Those all look fine. How about some results then?

```
summary(model1)
```

```
##
## Call:
## lm(formula = belief ~ CREDs, data = dat3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0406 -0.4328  0.1841  0.6448  1.9105
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.005256   0.025560   0.206    0.837
## CREDs       0.374365   0.025617  14.614   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9258 on 1310 degrees of freedom
## Multiple R-squared:  0.1402, Adjusted R-squared:  0.1395
## F-statistic: 213.6 on 1 and 1310 DF,  p-value: < 2.2e-16
```

Output is pretty similar to what you'd get from SPSS. Coefficients, standard errors, p-values. You can get confidence intervals for coefficients if so inclined.

```
confint(model1)
```

```
##                   2.5 %     97.5 %
## (Intercept) -0.04488658 0.05539805
## CREDs        0.32411144 0.42461915
```

8.3.1 Adding Predictors

Let's put the "multiple" in multiple regression and add some predictors. To predict belief from insecurity, CREDs, and reflection, you just add some plus signs.

```
model2 <- lm(belief ~ insecurity + CREDs + ref, data = dat3)

summary(model2)
```

```
##
## Call:
## lm(formula = belief ~ insecurity + CREDs + ref, data = dat3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2090 -0.4488  0.1549  0.6204  2.1858
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.005359   0.024897   0.215    0.830
## insecurity  -0.033148   0.025448  -1.303    0.193
## CREDs        0.386193   0.025147  15.358   <2e-16 ***
## ref         -0.217532   0.025559  -8.511   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9018 on 1308 degrees of freedom
## Multiple R-squared:  0.1855, Adjusted R-squared:  0.1836
## F-statistic: 99.27 on 3 and 1308 DF,  p-value: < 2.2e-16
```

BOOM! CREDs and reflection are significant predictors of belief. Insecurity, not so much.

8.3.2 Adding Interaction Terms

If you want to test for an interaction between two predictors, that's super easy to spell out in the formula. Here's a model looking for the interaction between CREDs and reflection:

```
model3 <- lm(belief ~ CREDs * ref, data = dat3)

summary(model3)
```

```
##
## Call:
## lm(formula = belief ~ CREDs * ref, data = dat3)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0388 -0.4686  0.1668  0.6164  2.5819
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.002031   0.024810  -0.082    0.935
## CREDs        0.390125   0.024868  15.688  < 2e-16 ***
## ref         -0.208348   0.024826  -8.392  < 2e-16 ***
## CREDs:ref    0.102531   0.024528   4.180 3.11e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.8964 on 1308 degrees of freedom
## Multiple R-squared:  0.1952, Adjusted R-squared:  0.1933
## F-statistic: 105.7 on 3 and 1308 DF,  p-value: < 2.2e-16
```

Hooray, a significant interaction! Note that it automatically includes all the lower-order predictive relationships for free when you include the interaction. There are various ways to decompose the interaction. There are even whole packages that people have written for that purpose. But it's beyond the scope of this introduction.

9 Closing

There was a quick and very rough intro to R. I'll close with what to do if you get lost or confused. Here are the 4 steps I'd recommend (and do all the time):

1) Borrow someone else's code and adapt it for what you're doing.
2) Learn to re-use and streamline often-used code chunks in your workflow. You'll do some tasks (basic wrangling) on almost ever dataset you analyze. So don't reinvent the wheel each time.
3) Confused? Check Navarro.
4) Google is your friend.