R live- day 1

Marc Kissel

May 25, 2018

Intro section

Why use R

it is more than just a way to run statistics. You can make nice figures, run computer simulations, create interactive apps, and even make a website!

I don't know where i read this analogy, but it helps to think of R compared to SPSS, Excel, etc as looking at the difference between taking a bus and driving a car. Taking a bus is easy and carefree & it gets you from point A to point B. But there is not much freedom in where you go. With a car you can go anywhere, but you have to learn how to drive and pay attention during the trip. In R you can do almost anything with your data. But it takes some time and effort…though i think it is worth it!

How to get help

No one knows everything about R. I often Google the same issue many times since i forget. A great place to look for help is Stack Overflow. If you are on Twitter, lots of great folks follow the #Rstats. Often, I find the best way to Google it is to take the thing you want to know and add "rstats" to the end or the name of the package i'm using. So, if i wanted to change the background color of a plot "ggplot2 background color" brings up a link to a stackoverflow discussion on it.

Also, most packages have help online. some even have tutorials. I like to run through the online tutorial of a new package since it helps me to figure out what is going on.

A short warning: if you ask a question be prepared to give details on your question. It might be better to ask me first and if i don't know the answer i can help you get your question into a format that allows others to help you!

Working with R

The *console* is where commands are run. You can write something in the console to speak directly to R, or write in the *source* and then send the script to the console. You can recognize the console since every lines starts with a '>'

go to the console and type the following and then hit enter

```r
7 + 6
```

```
## [1] 13
```

you should see something like this:

> 7 + 6 [1] 13

note the [1], which tells you the line number. This becomes useful later on. Also note it told you that 7 + 6 = 13. congrats! you just turned your expensive computer into a $5 calculator

Short overview of how R works

One way that i think about R that helps me to work it is to remember that everything in R is an "object." What this means is that in R we make objects (or we assign values to objects) and then do things to these objects. for example check out the code below

```r
a <- 4
```

What this says is that we are going to create the object '**a**' and then assign the value '4' to it. If you run the code in R Studio you will see that the 'environment' panel now lists 'a' and gives its value.

The arrow shortcut key for a pc is "alt" + "-"". For a mac it is"Option" + "-"

```r
b <- "my name"
c <- FALSE
d <- c(1,2,3.0,4)
```

protip: When you assign values to objects, be aware that not all names are possible. R reserves some names for special functions. here are the rules

1. Can use letters/numbers, But has to start with letter (impt for when you import data).
2. no spaces, so use CamelCase or Snake_case

to store more than one element in an object we need to remember a trick

if you want to make a series of numbers, characters, whatever, you need to use a special sequence. to do this, we must combine the values we want to assign with the c function, which combines values into a vector or list

```r
my_list <- c(1,2,3,4,5)
my_list_2 <- c("my_name", "my_address", "my_number")
```

Using objects

Once you assign *something* (e.g., a value, a list of numbers, a character vector) to a object you can interact with that object. We do this by using what is called a **function**. A function is a built-in R command that does things to objects. Functions take what we call **arguments**.

To call a function in R you write the name of the function and then put the arguments of that function in parenthesis. Each argument is separated by a comma. Some functions have one argument but many have more. Often, the arguments have default values which means you don't have to specify what that is. For example the **mean()** function defaults to na.rm = False, which means it won't automatically remove missing values. Oftentimes you might want to skip the missing values, so you would add na.rm = TRUE

R read "NA" as standing for missing values. However, it is counted as an entry. so if you have NA values you need to remember to tell R to skip those if calculating the mean

```r
sum(d)
```

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

```r
mean(d)
```

```
## [1] 2.5
```

```r
mean(d, na.rm = FALSE) #doesn't change result since we have no NAs yet
```

```
## [1] 2.5
```

```r
e <- c(1,2,3,4,NA,5)
mean(e)
```

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

```r
mean(e, na.rm = TRUE)
```

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

```r
seq(from = 1, to = 10, by =2)
```

```
## [1] 1 3 5 7 9
```

if you need help on a function, you can type a question mark before its name like this: ?mean()

The way in which you can interact with the object is based on the *class* of the object. The class of the object is assigned by R based on what it thinks the object is supposed to be.

```r
class(a)
```

```
## [1] "numeric"
```

```r
class(b)
```

```
## [1] "character"
```

```r
class(c)
```

```
## [1] "logical"
```

```r
class(d)
```

```
## [1] "numeric"
```

In this case, the function is ‘class’ and the argument is the object whose class you want to know.

Class

A **class** defines what kinds of operations can be implemented on an object & how a function will return a value. It is important to keep track of the classes of your objects. Class mistakes are probably the most common kind of problem in

Once you have an object stored, you can interact with it. but remember that the way you interact with it via functions depends on its class. . . .

```
length(my_list)
```

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

```
length(my_list_2)
```

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

```
sum(my_list)
```

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

```
my_list[4] #gets the 4 value.
```

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

```
#sum(my_list_2) #this kicks back an error
```

running the second line gives you a message like this:

"Error in sum(my_list_2) : invalid 'type' (character) of argument"

What this means is that the 'type' or 'class' of the object is not something that the sum function knows what to do with. **Learning how to read R errors is useful.**

If you don't know what the error means, copying part of it and searching Google is a pretty good way to find help.

(*short digression: one of the things about R that makes it easier to use is that many of the functions work by first checking out the class of the object you are asking it to work on. What this means is that the same function can be applied to different types of classes. This doesn't really affect much now, but as you investigate R more you can see how this works.*)

Starting a project

1. best advice i can give is that you want everything to be in one place (makes things a bit harder at first)

2. this is what R Projects are for

3. start a new project and store all your data there

4. when you start/open a project, it sets R's working directory to that folder. this way, you can easily locate things

Packages

R comes with many functions already instaled. But what makes it super powerful is that people can add to these functions by writing their own (and we will write our own function in week 7!). A bunch of functions bundled together is called a **Package**. You are going to find lot of useful packages as you us R. To install a package in R we use the function "install.packages," which has the argument the name of the package.

install.packages("name_of_package") #install a package library(name_of_package) #load the package for your session

```
#install.packages("tidyverse")
```

you only have to install a package once, but **everytime you reopen R you have to reload the pacakage with the libr**

```
library(tidyverse)
```

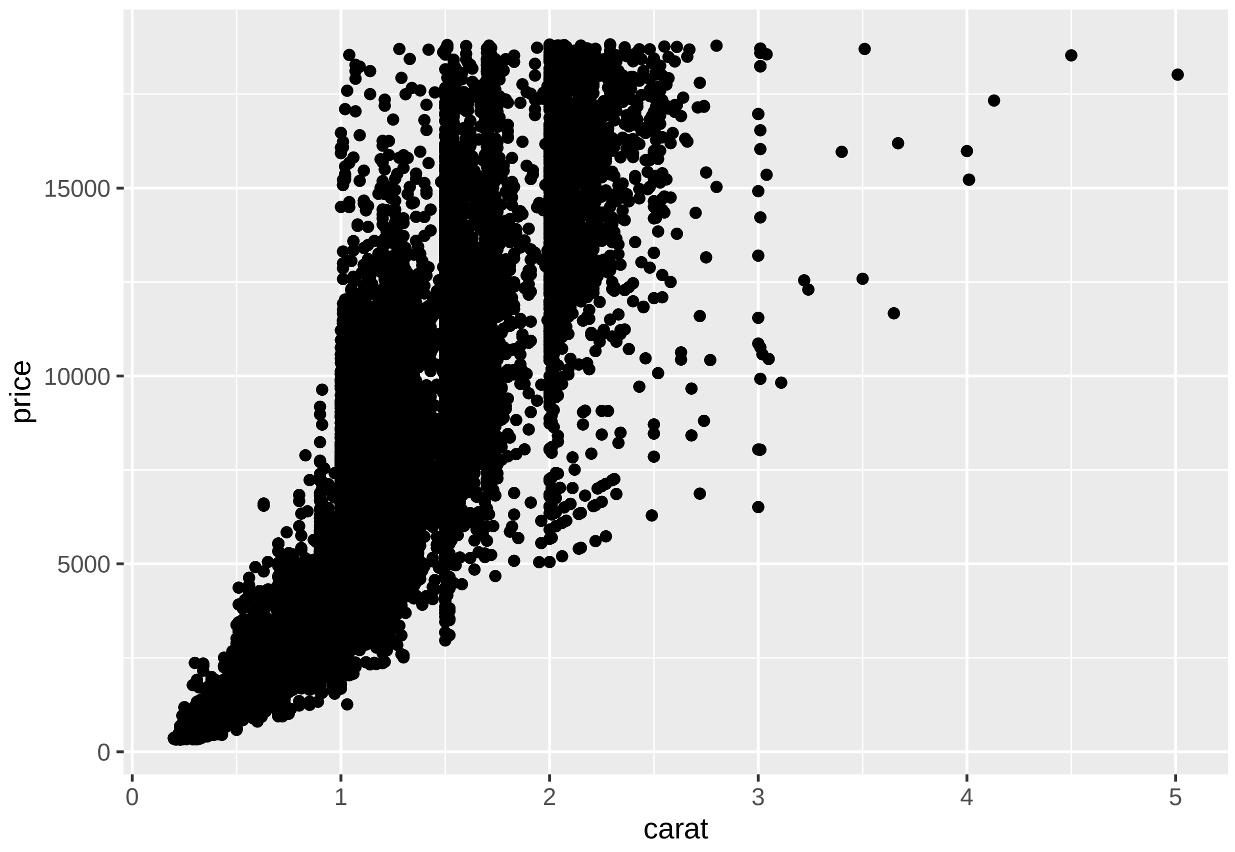
```
## Warning: package 'tidyverse' was built under R version 4.0.3

## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

In the case of this package, R tells us what is being loaded and also that there are some conflicts. We can usually ignore these, but basically it just says that some functions from this new package have the same name as functions from the base R package. Good to pay attention to but usually not a huge problem unless you are doing a lot of things in one script

Here is a short piece of code that might make some sense. It calls a function ggplot (part of the tidyverse package), and then makes a plot. Then the ggsave function saves the new plot as a pdf in the working directory of the project folder.

the last line takes the data set that was used (diamonds) and write as CSV file for that.

```
ggplot(diamonds, aes(carat, price)) +
  geom_point()
```



```
ggsave("diamondPlot.pdf")
```

```
## Saving 6.5 x 4.5 in image
```

```
write_csv(diamonds, "diamonds3.csv")
```

How to get data into R (updated!)

Probably the hardest part of starting to use R is getting your data into R and then being able to work with it fluidly. This has gotten easier over time but still can be tricky. For me, the biggest hurdles are

1. getting the file in a good format (which is next week's work!)
2. making sure that when it is read into R all of the data is being read correctly by R

Using readr

The readr package is part of the tidyverse package. One of the points of the tidyverse is to try to make things consistent.

Here are some of the functions to read data into R

read_csv - comma separated values

read_csv2() - semicolons

read_fwf() - fixed width

For all of these the first argument is the file name. Other arguments can be used to if need be. For example, use "col_names = FALSE" if no headers to your data (uncommon)

What is nice about read_csv etc is that it tells you what it **thinks** that class of each column is

```
#New_data <- read_csv("diamonds.csv")
```

You can see that it is reading the file and trying to 'guess' what the right type of class the data is. For more on this see:

https://readr.tidyverse.org/ or http://r4ds.had.co.nz/data-import.html

You can also use the 'import data set' tool which lets you 'point and click' and then generates the code. This is helpful and probably something i should use more often

Things to be aware of when importing data into R

As we talked about, the biggest problem is dealing with factors. factors are categorical data. In the old days R used to import any string as a factor. This caused some problems! Now, readr fixes this. But we still need to be aware of how factors work. The following code, taken mostly from http://www.ats.ucla.edu/stat/r/modules/factor_variables.htm gives a workthough on what factors are and how to think about them.

We are going to start by making a new object and then turning it into a factor. then play around with this new factor to see how it works

###Factors

```
set.seed(100) #this sets the randomizer so if you run this code you get the same results i did
temp <- sample(0:1, 20, replace=T)
temp
```

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

```
is.factor(temp)
```

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

```
is.numeric(temp)
```

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

```
#ok, so we now have a list of numbers
#remember that factors are categories
#so, we are going to assign factors to these values
#uses the argument labels
temp_f <- factor(temp, labels = c("femur", "tibia"))
temp_f #hey, that is kinda cool! lets check if it still numeric
```

```
##  [1] tibia femur tibia tibia femur femur tibia tibia tibia femur tibia tibia
## [13] tibia tibia femur tibia tibia femur femur femur
## Levels: femur tibia
```

```
is.numeric(temp_f)
```

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

```
is.factor(temp_f) #sweet!
```

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

```
#so that works with numbers, but what about strings

size <- c("small", "medium", "small", "small", "small", "small", "medium", "small",
          "medium", "medium", "medium", "medium", "medium", "tall", "tall",
          "small", "medium", "medium", "small", "tall")
is.factor(size)
```

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

```
is.character(size)
```

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

```
#ok, so how do we turn this into a factor? lets what we did before
size_f <- factor(size)
is.factor(size_f)
```

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

```
size_f #but lets take a closer look at the levels..
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## Levels: medium small tall
```

```
?levels #read the help here and try to see what it does
```

```
## starting httpd help server ... done
```

```
levels(size_f)
```

```
## [1] "medium" "small"  "tall"
```

```r
# ok, so notice the order of the levels. here, they are sorted alphabetically
#this might not be what you want (in fact, it rarely is.). I've had many
#issues with R that comes down to this problem. Dplyr/ggplot makes it a little
#easier, but you still going to have problems
#so, how to fix this
#we need to set the levels!
size_f2 <- factor(size, levels=c("small", "medium", "tall"))
is.factor(size_f2)
```

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

```r
levels(size_f2) #woot. they are now in the right
```

```
## [1] "small"  "medium" "tall"
```

```r
###
#sometimes you might need to use ordinal data. for that we use ordered factors
#note: ordinal data is when we the order of the values matters,
#but we can't say how much of a difference there is between each.
#the data is ranked but the distance between categories isn't clear
#so, if we are measuring something like job satisifaction
#hard to say how much better 'very happy' is from 'happy'
#e.g. likert scale

size_ordered <-  ordered(size, levels =c("small", "medium", "tall"))
size_ordered
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## Levels: small < medium < tall
```

```r
size_f2
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## Levels: small medium tall
```

```r
#note in size_ordered you see the '<'

table(size)
```

```
## size
## medium  small   tall
##      9      8      3
```

```r
table(size_ordered)
```

```
## size_ordered
##  small medium   tall
##      8      9      3
```

```r
#the last thing i want to mention is adding data. now,
#you might think this is easier to do in excel
#but remember that R has the plus of perserving
#all you steps. Plus, this way you aren't messing with
#the original data

size_f2[21] <- "very.tall"
```

```
## Warning in `[<-.factor`(`*tmp*`, 21, value = "very.tall"): invalid factor level,
## NA generated
```

```r
#we get an error message. note it says 'invalid factor'
size_f2
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## [21] <NA>
## Levels: small medium tall
```

```r
#there is now a NA. this is cause R doesn't know how to deal
#with the new factor
#we need to add the new factor
size_f2 <- factor(size_f2, levels = c(levels(size_f2), "very.tall")) #ok, so this
#is a bit long. but levels is the same. just cheating a bit by usinf the levels(size_f2)
#function to get the original levels from the list. As you getbetter w/ R
#you learn these little tricks to help. avoids making mistakes...
size_f2
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## [21] <NA>
## Levels: small medium tall very.tall
```

```r
size_f2[21] <- "very.tall"
####ok, but lets say we don't want 'very tall' anymore. This could become
#problem when making images since it may still show up even if we just
#remove it
#first, we get rid of the elements we dont want
size_f2_new <- size_f2[size_f2 != "very.tall"]
#then, we 'refactor' it
size_f2_new <- factor(size_f2_new)
size_f2_new
```

```
##  [1] small  medium small  small  small  small  medium small  medium medium
## [11] medium medium medium tall   tall   small  medium medium small  tall
## Levels: small medium tall
```

```r
#note the difference between the 2 tables
table(size, temp)
```

```
##         temp
## size     0 1
##   medium 3 6
##   small  3 5
##   tall   2 1
```

```
table(size_f2_new, temp_f)
```

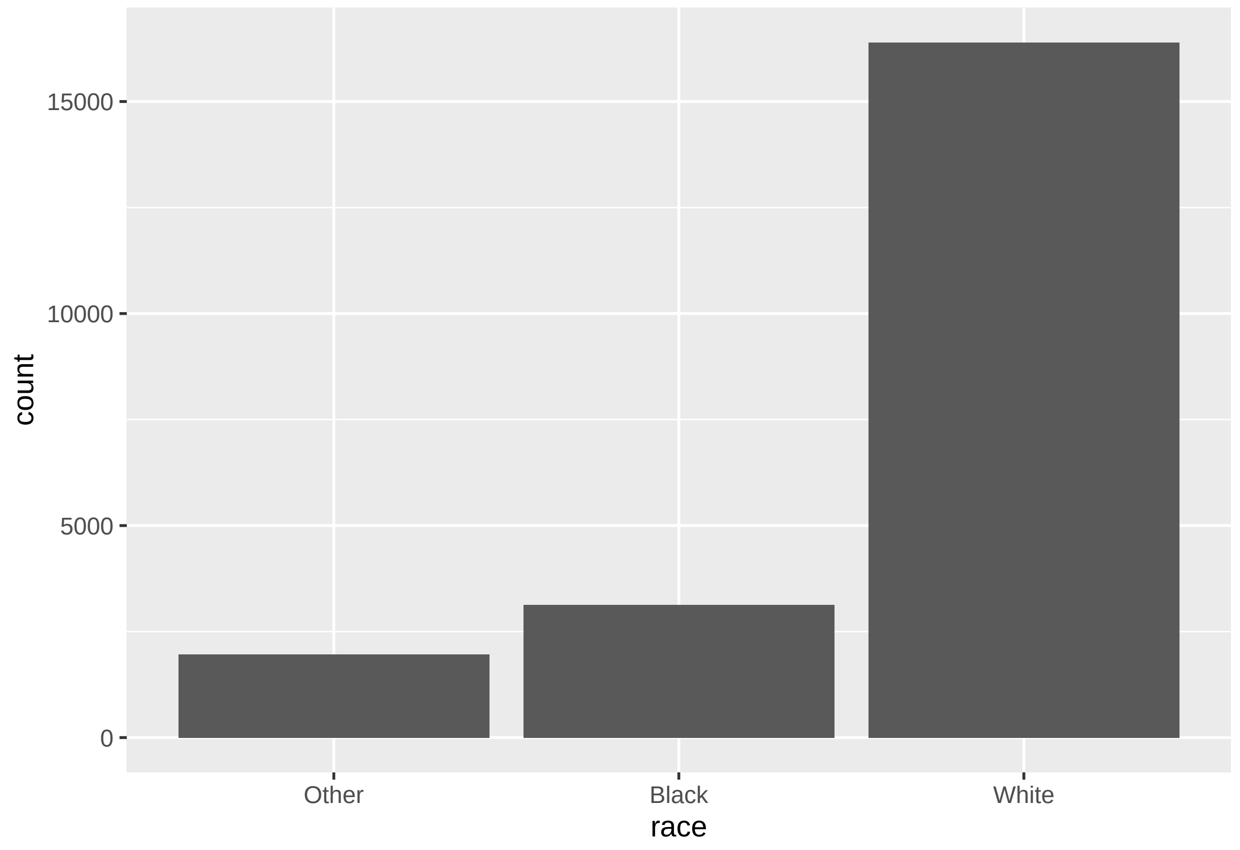
```
##            temp_f
## size_f2_new femur tibia
##      small      3     5
##      medium     3     6
##      tall       2     1
```

```
#finally, lets put it all together
data <- c(34, 39, 63, 44, 47, 47, 57, 39, 48, 47, 34, 37, 47, 47, 39, 47, 47, 50, 28, 60)

my_data <- data.frame(temp, temp_f, size, size_f2_new, data)
my_data
```

```
##    temp temp_f   size size_f2_new data
## 1     1  tibia  small       small   34
## 2     0  femur medium      medium   39
## 3     1  tibia  small       small   63
## 4     1  tibia  small       small   44
## 5     0  femur  small       small   47
## 6     0  femur  small       small   47
## 7     1  tibia medium      medium   57
## 8     1  tibia  small       small   39
## 9     1  tibia medium      medium   48
## 10    0  femur medium      medium   47
## 11    1  tibia medium      medium   34
## 12    1  tibia medium      medium   37
## 13    1  tibia medium      medium   47
## 14    1  tibia   tall        tall   47
## 15    0  femur   tall        tall   39
## 16    1  tibia  small       small   47
## 17    1  tibia medium      medium   47
## 18    0  femur medium      medium   50
## 19    0  femur  small       small   28
## 20    0  femur   tall        tall   60
```

Factors in the tidyverse (see http://r4ds.had.co.nz/factors.html for more on working with factors in R)

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



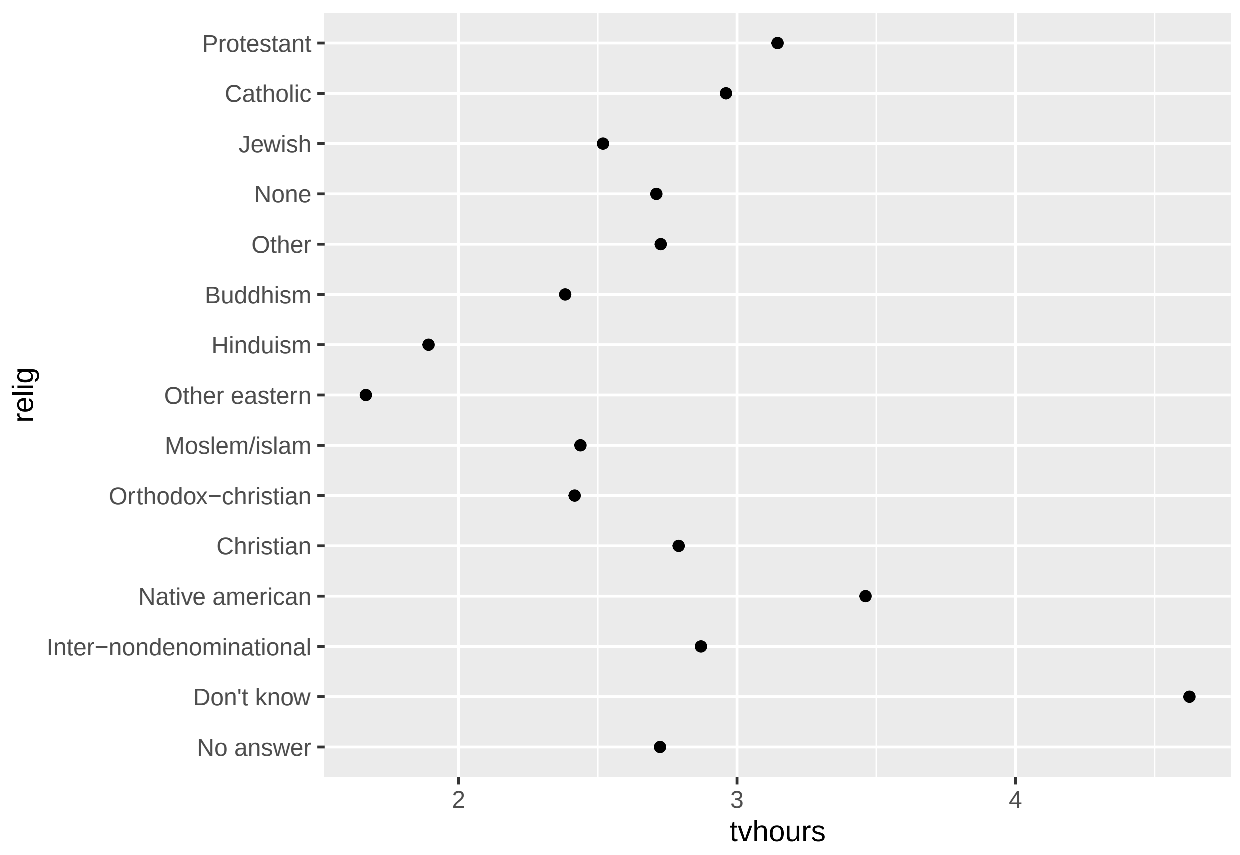
1. reorder factors. Often when making plots we want to change the order they appear in. One was is shown above. Another way is to use the forcats package that comes with the tidyverse

lets set the data up:

```
relig_summary <- gss_cat %>%
  group_by(relig) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



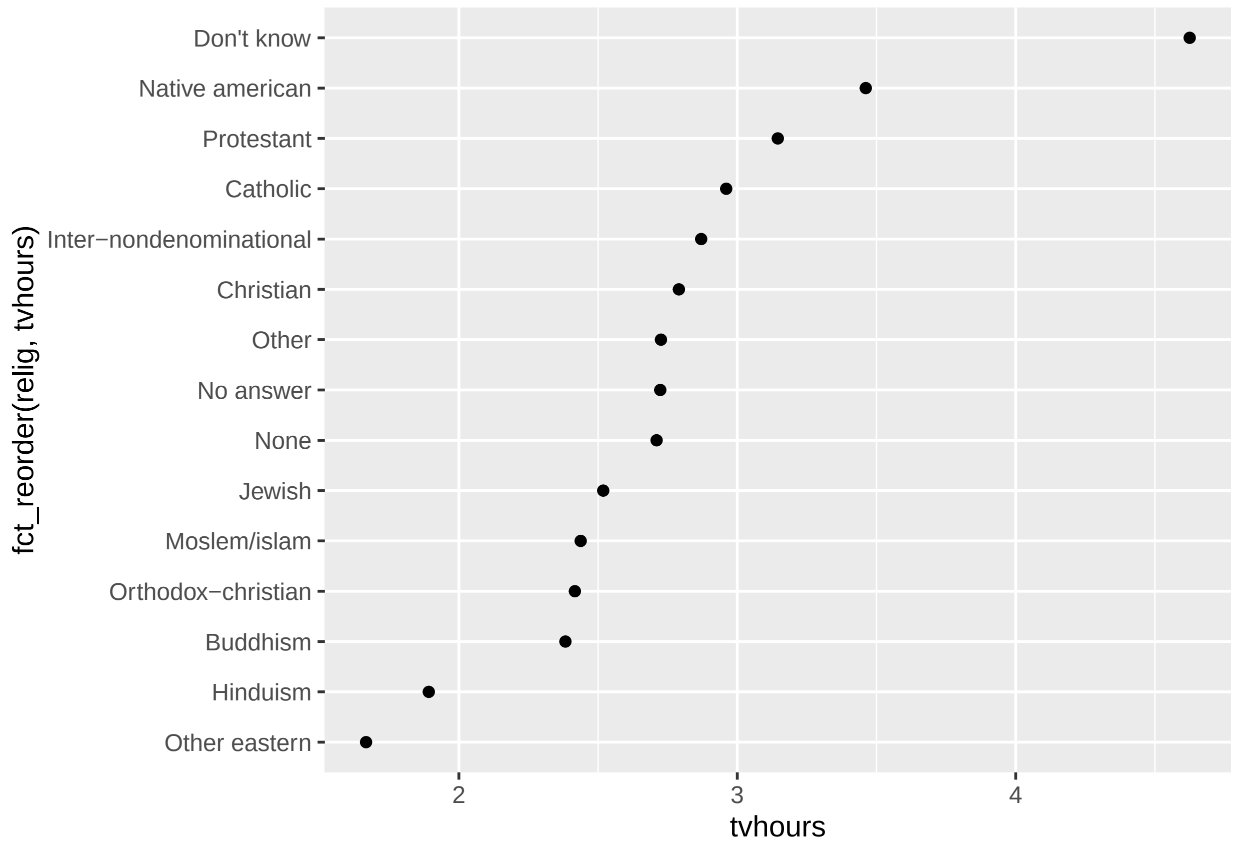
when we plot this the factors are not organized by tv hours, which makes it hard to examine.

We can improve it by reordering the levels of relig using fct_reorder().

fct_reorder() takes 2 main arguments:

1. the factor we want to reorder
2. the numeric vector we want to reorder by

```
ggplot(relig_summary, aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point()
```



rename factors In the Tidyverse it is easy to rename factors

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
                              "Republican, strong"    = "Strong republican",
                              "Republican, weak"      = "Not str republican",
                              "Independent, near rep" = "Ind,near rep",
                              "Independent, near dem" = "Ind,near dem",
                              "Democrat, weak"        = "Not str democrat",
                              "Democrat, strong"      = "Strong democrat"
  )) %>%   count(partyid)
```

```
## # A tibble: 10 x 2
##    partyid                   n
##    <fct>                 <int>
##  1 No answer               154
##  2 Don't know                1
##  3 Other party             393
##  4 Republican, strong     2314
##  5 Republican, weak       3032
##  6 Independent, near rep  1791
##  7 Independent            4119
##  8 Independent, near dem  2499
##  9 Democrat, weak         3690
## 10 Democrat, strong       3490
```

More on the Tidyverse

Next week i'll do more on this. . . ..

#Vectors and matrices. ###

#Dealing with single values or functions is easy enough. The #thing that trips most people up is the dreaded list.

#Technically, just about every object in R is some kind of #a list. We'll start out with some simple objects and #figure out how to index them, manipulate them, and perhaps #most importantly, how we can visualize them.

#We'll start out with a vector. We'll even use an informative #name to keep things clear.

example_vec <- rnorm(20)

example_vec

#This is a vector of 20 draws from a random normal with #a mean of 0 and a standard deviation of 1.

#The thing you need to know most about vectors is that #you are able to index them. Use square brackets to specify #that you are indexing. Since a vector is one dimensional, #you only need a singe index.

#Let's say you wanted the first entry.

example_vec[1]

#What about the third through 6th entry?

example_vec[3:6]

#Remember that you read ":" as "through".

#These are the easy ones, but there are much more #powerful techniques to bend vectors to your will.

#Let's say you wanted the fourth, fourteenth, and #eleventh entries, in that order. To do this, you #have to use the c() function, which you read as #"concatenate", to bind together a series of numbers #into a vector that you then pass as an index for #the example.vec vector.

example_index <- c(4,14,11)

example_vec[example_index]

#Check the original vector to see what we just did. #A quick tip to make a vector readable in terms of #the order of its entries is to turn it into a matrix

as.matrix(example_vec)

#This makes life really easy when you are trying # to quickly find things in a busy workspace or in #a big data frame.

#Another indexing trick is exclusive indexing which #let's you indicate which entries you DON'T want #to recover from a vector. Use a minus sign in #front of the index to do this. So, if you did not #want the 10th and 13th entries in our vector, you #would do this:

example_vec[-c(10,13)]

#You'll note that I didn't make an entirely new #vector in the workspace. I just called c() inside #the square brackets. This is sometimes easier to #read and it can keep the workspace uncluttered.

#That about covers all you need to know about #indexing and manipulating vectors. But, what #are you really doing?

#The really tricky part of all this is visualizing #what is going on. You'll want to come up with some #visual metaphors for what is going on. In my own #case, when I was learning how to do this, I imagined

#different configurations of lego bricks from my #childhood. Sounds stupid, but if it's stupid and it #works, it isn't stupid.

#I can't emphasize this enough: #DRAW OUT YOUR OBJECTS AND COME UP WITH A PHYSICAL #METAPHOR FOR OBJECTS IN YOUR WORKSPACE. #It will save you tons of grief.