notes from workshop

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. but 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

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 pacakge since it helps me to figure out what is going on.

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)
```

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.

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 R

some class types

lgl stands for logical, vectors that contain only TRUE or FALSE. int stands for integers. can't take decimals dbl stands for doubles, or real numbers. chr stands for character vectors, or strings. str stands for string....text factor are categorical data

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

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

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 whatthe 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. but what makes it super powerful is that people can add to these functions by writing their own. a bunch of functions bundled together is called a package. You are going to find lot of useful ones. 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

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

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

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------

## v ggplot2 2.2.1     v purrr   0.2.4
## v tibble  1.3.4     v dplyr   0.7.4
## v tidyr   0.7.2     v stringr 1.2.0
## v readr   1.1.1     v forcats 0.2.0

## -- 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

as a rule, it is best to load all the packages you are using at the beginning of your R script. In other words, if i am starting a new .r file i usually do it like this

.# line 1 #my_new_r_script. # the name of the file line 2 library(tidyverse) line 3 library (MASS)

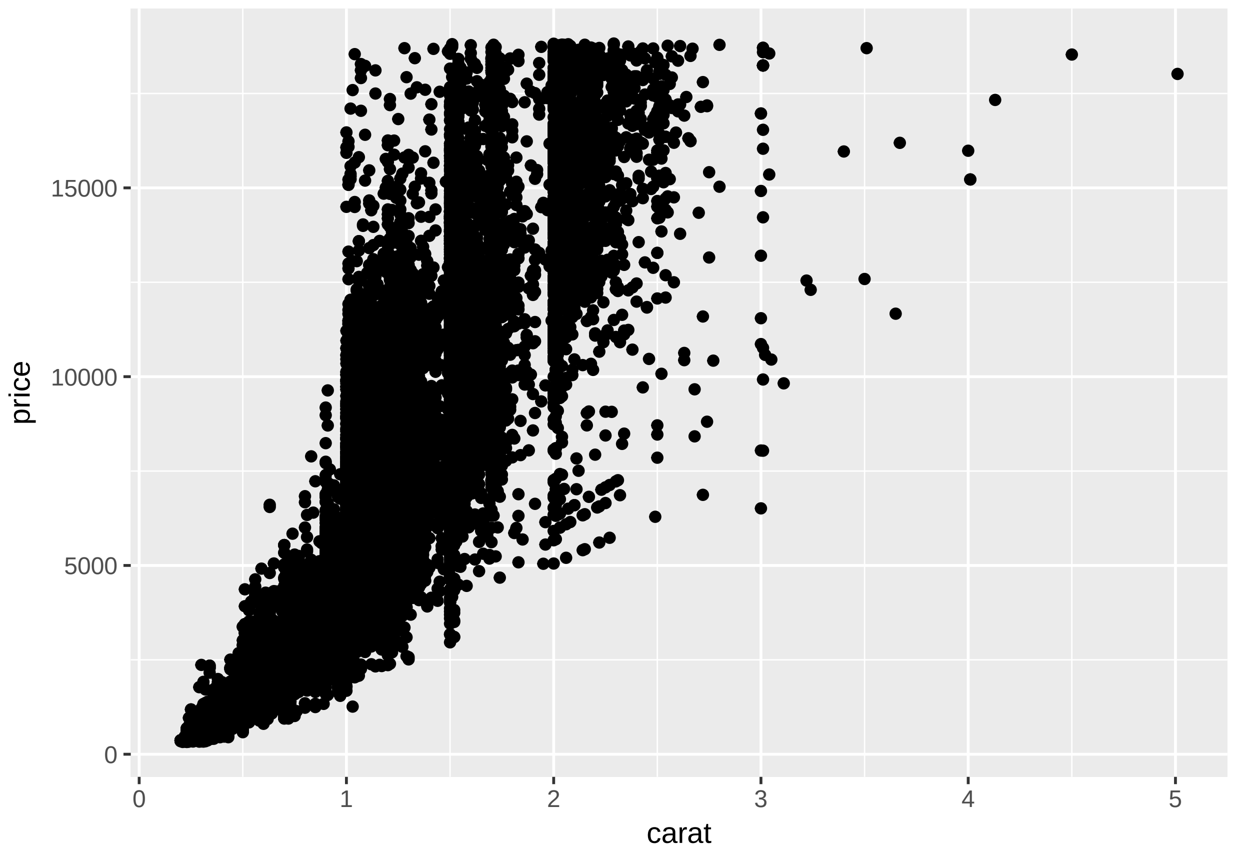
This just keeps things clean and easy to read.

packages I used in this workshop Be sure to install all of these:

1. tidyverse
2. gapminder
3. scales
4. nycflights

Here is a short piece of code that should make some sense. it calls a function ggplot (part of the tidyverse package), and then makes a plot. then ggsave 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.

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



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

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

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

A quick example of some code

This might make some sense based on last week. i wrote this to see if i could recreates the plot from the Oka et al. paper. it is a bit complex and probably could be simplified a bit

```
library(tidyverse)

war <- read_csv("PopSizeTemp.csv")
```

```
## Warning: Missing column names filled in: 'X13' [13]
```

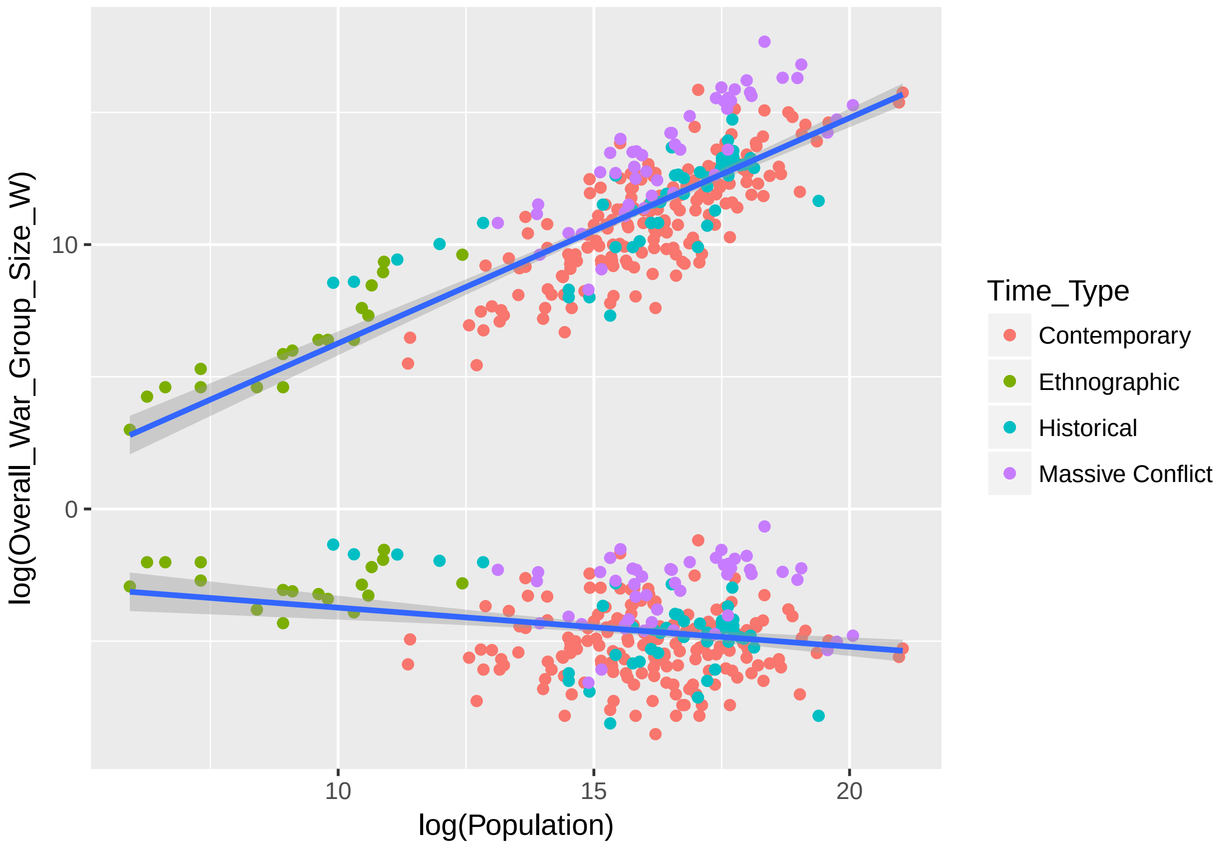
```
## Parsed with column specification:
## cols(
##   Society = col_character(),
##   Year = col_integer(),
##   Type_of_Society = col_character(),
##   Time_Type = col_character(),
##   Military_Status = col_integer(),
##   Population = col_integer(),
##   Overall_War_Group_Size_W = col_integer(),
##   `X_(W)` = col_double(),
##   LNP = col_double(),
##   LNW = col_double(),
##   W.P = col_double(),
##   `X (W)` = col_double(),
##   X13 = col_double()
## )
```

```
plot_Fig1a1 <- ggplot() + geom_point(data = war, aes(x= log(Population), y = log(Overall_War_Group_Size_
```

```
## Warning: Ignoring unknown aesthetics: text
```

```
## Warning: Ignoring unknown aesthetics: text
```

```
plot_Fig1a1
```



```
DCI <- read_csv("X_Factor_Dataset.csv")
```

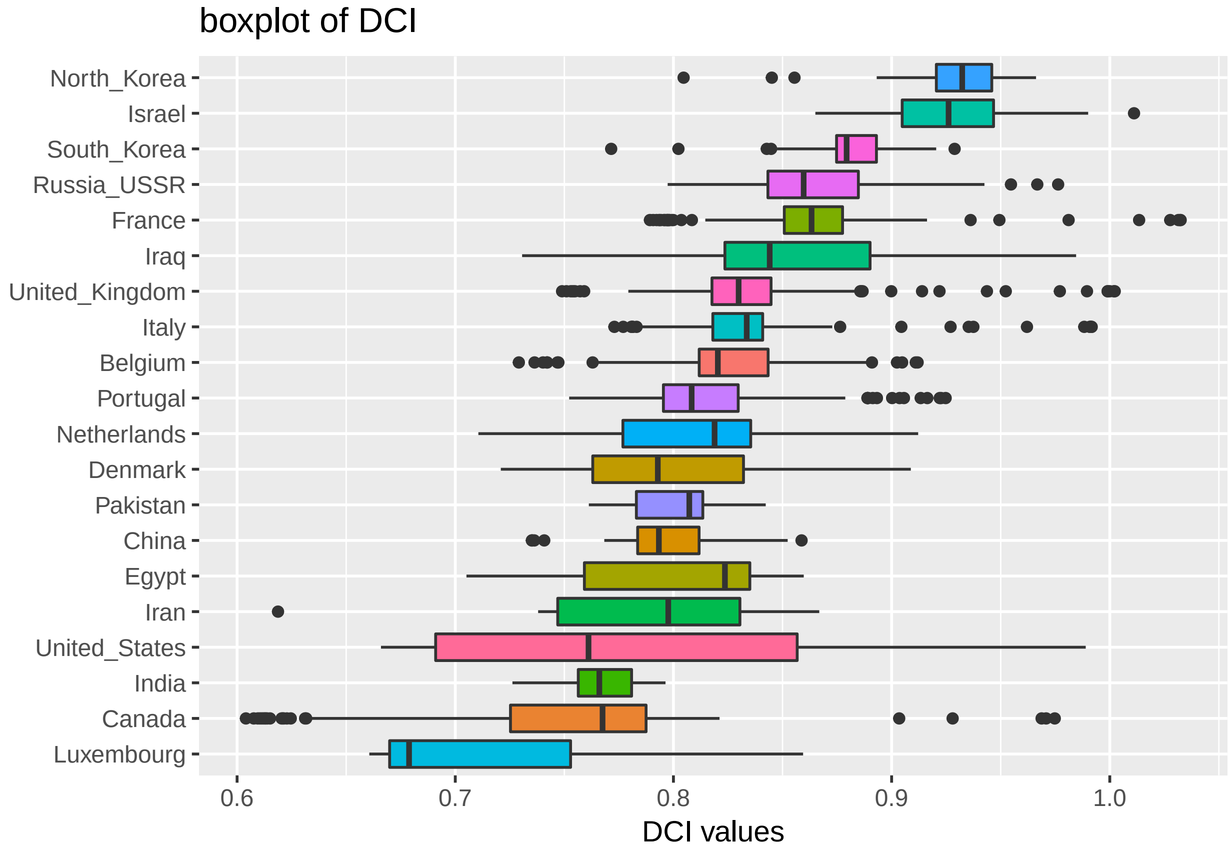
```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   Year = col_integer()
## )
```

```
## See spec(...) for full column specifications.
```

```
DCI_interact <-  DCI %>% gather(value = X, key = country, Belgium:Iraq)
DCI_summary <- DCI_interact %>% group_by(country) %>% summarise(DCI_mean = mean(X, na.rm = TRUE), DCI_s
p <- ggplot(data=DCI_interact, aes(x=reorder(country, X, na.rm=TRUE), y = X, fill= country))
```

```
DCI_box <- p + geom_boxplot() +labs(x=NULL, y = "DCI values", title = "boxplot of DCI") +coord_flip() +
DCI_box
```

```
## Warning: Removed 1234 rows containing non-finite values (stat_boxplot).
```



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 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 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")
```

```
## Parsed with column specification:
```

```
## cols(
##   carat = col_double(),
##   cut = col_character(),
##   color = col_character(),
##   clarity = col_character(),
##   depth = col_double(),
##   table = col_double(),
##   price = col_integer(),
##   x = col_double(),
##   y = col_double(),
##   z = col_double()
## )
```

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] 0 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 0 0 0 1
```

```
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] femur femur tibia femur femur femur tibia femur tibia femur tibia
## [12] tibia femur femur tibia tibia femur femur femur tibia
## 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"
```

```
# 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
```

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

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

```
###
#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
```

```r
#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
```

```
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
```

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

```
##         temp
## size     0 1
##   medium 5 4
##   small  6 2
##   tall   1 2
```

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

```
##            temp_f
## size_f2_new femur tibia
##      small      6     2
##      medium     5     4
##      tall       1     2
```

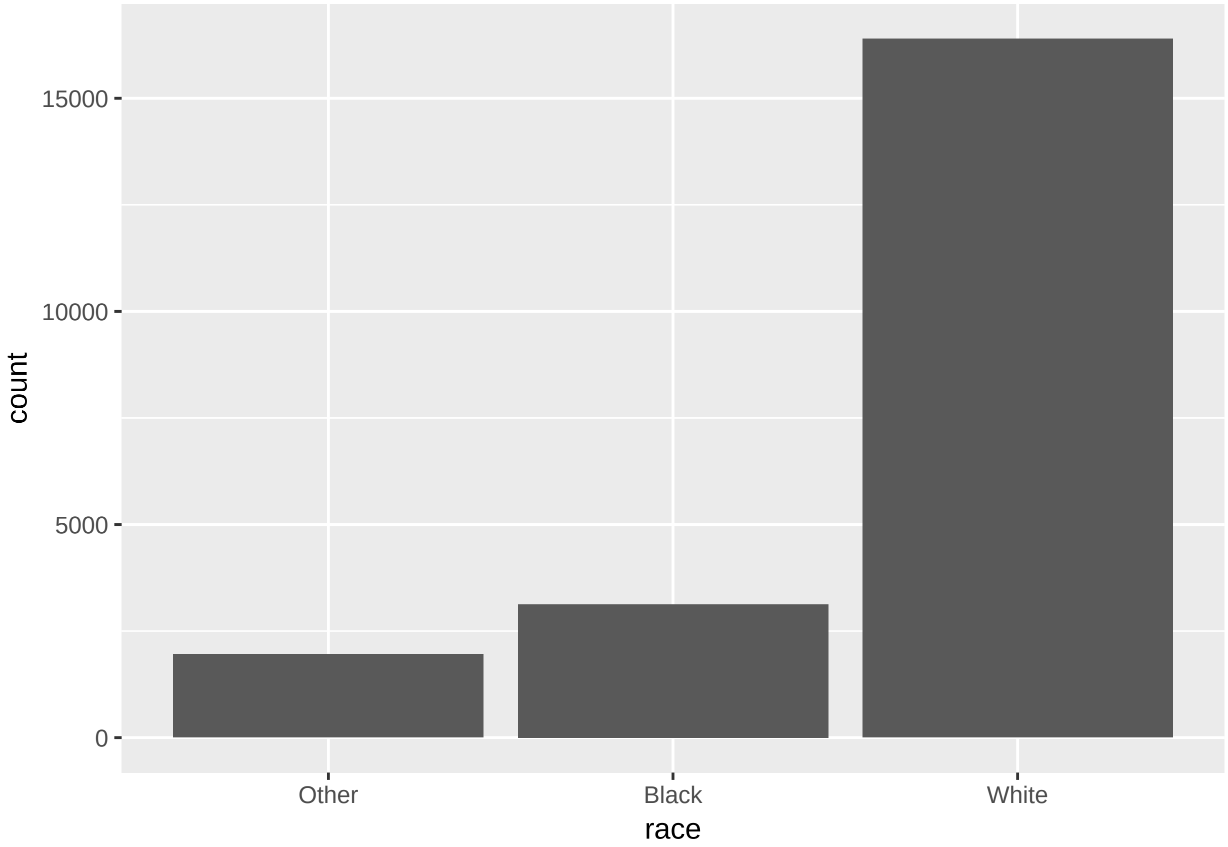
```
#finally, lets put it all together
data <- c(34, 39, 63, 44, 47, 47, 57, 39, 48, 47, 34, 37, 47, 47, 39, 47, 47, 50, 28, 60)

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

```
##    temp temp_f   size size_f2_new data
## 1     0  femur  small       small   34
## 2     0  femur medium      medium   39
## 3     1  tibia  small       small   63
## 4     0  femur  small       small   44
## 5     0  femur  small       small   47
## 6     0  femur  small       small   47
## 7     1  tibia medium      medium   57
## 8     0  femur  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    0  femur medium      medium   47
## 14    0  femur   tall        tall   47
## 15    1  tibia   tall        tall   39
## 16    1  tibia  small       small   47
## 17    0  femur medium      medium   47
## 18    0  femur medium      medium   50
## 19    0  femur  small       small   28
## 20    1  tibia   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()
```



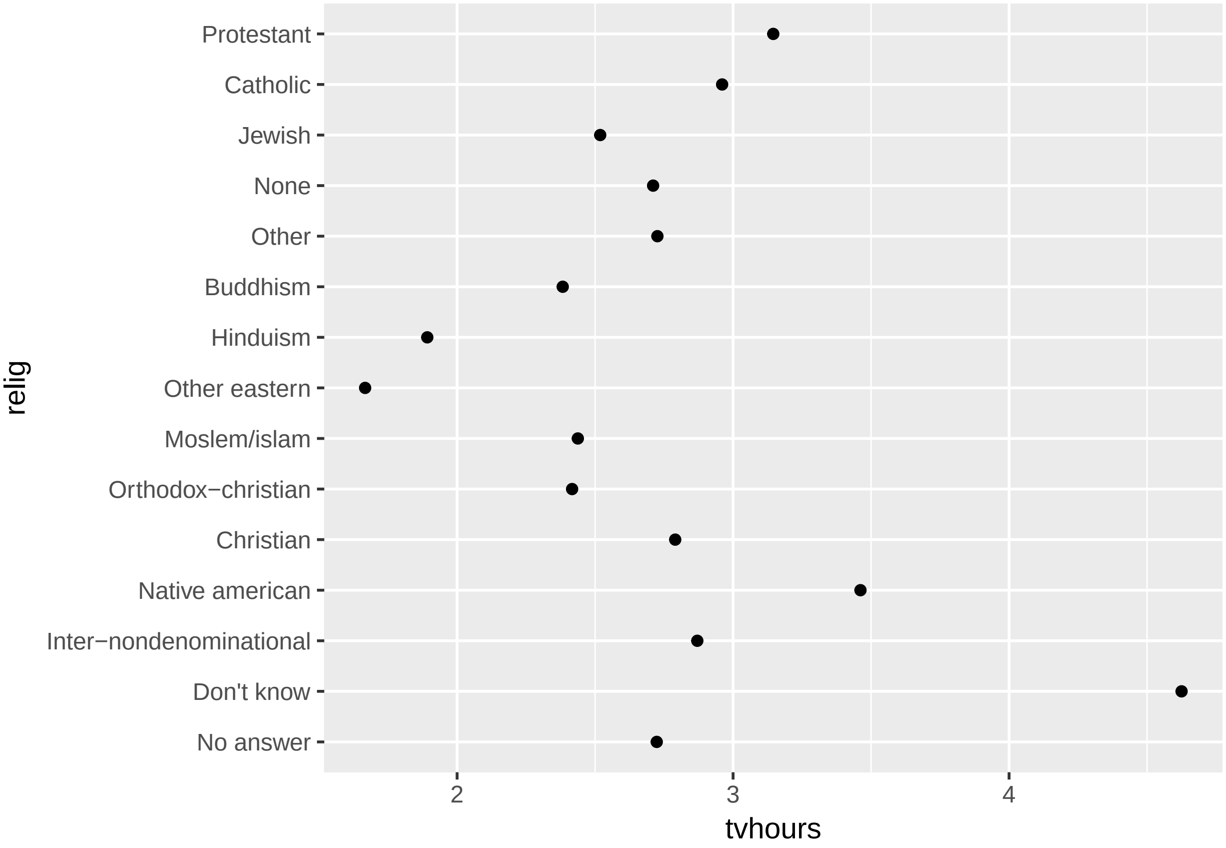
reorder factors.

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()
  )
```

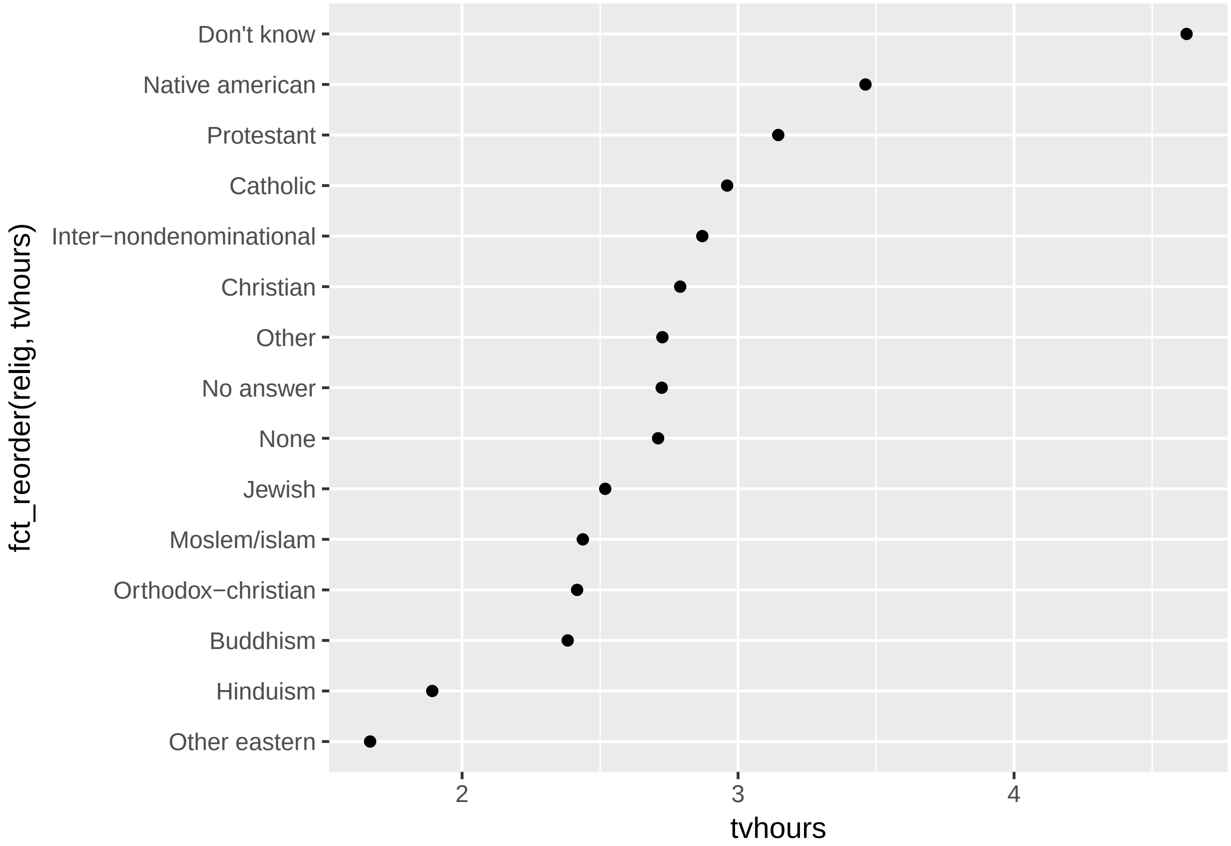
```
ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



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 A few 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()
```



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
##                   <fctr> <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
```

Day 2: data wrangling in R

note: a lot of this i got from chpts from the R for Data Science book......

data transformation is key to R. We transform data to do new stuff using parts of the the Tidyverse package (specifically the dplyr package.

Below are the libraries i used for this section

```r
library(tidyverse)
library(nycflights13) #might need to install this package
```

```
## Warning: package 'nycflights13' was built under R version 3.4.4
```

we are going to use the nycflights 13 dataset since it is large and gives examples of how to transform and work with data.

In R, a collection of variables is called a **dataframe**. In the tidyverse, they call the same thing a **tibble**. mostly there is not much of a difference but sometimes you want to view data as a tibble since it looks nicer in R. you can use the function as_tibble to turn a basic dataframe into a tibble

to look at the data in dataframe/tibble there are different functions. each has it benefits. glimpse(), from the tidyverse, makes sure that when you run that code everything lines up in the console window.

```r
str(flights) # old R
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

```r
glimpse(flights) # Dplyr package from the tidyverse
```

```
## Observations: 336,776
## Variables: 19
## $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
## $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
## $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
## $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
## $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
## $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
## $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
## $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
## $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
## $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
## $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
## $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
## $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
## $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
## $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
```

```
print(flights) #first 10
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
print(flights, n=40) #first 40
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## 11  2013     1     1      558            600        -2      849
## 12  2013     1     1      558            600        -2      853
## 13  2013     1     1      558            600        -2      924
## 14  2013     1     1      558            600        -2      923
## 15  2013     1     1      559            600        -1      941
## 16  2013     1     1      559            559         0      702
## 17  2013     1     1      559            600        -1      854
## 18  2013     1     1      600            600         0      851
## 19  2013     1     1      600            600         0      837
## 20  2013     1     1      601            600         1      844
## 21  2013     1     1      602            610        -8      812
## 22  2013     1     1      602            605        -3      821
## 23  2013     1     1      606            610        -4      858
## 24  2013     1     1      606            610        -4      837
## 25  2013     1     1      607            607         0      858
## 26  2013     1     1      608            600         8      807
## 27  2013     1     1      611            600        11      945
## 28  2013     1     1      613            610         3      925
## 29  2013     1     1      615            615         0     1039
## 30  2013     1     1      615            615         0      833
## 31  2013     1     1      622            630        -8     1017
## 32  2013     1     1      623            610        13      920
## 33  2013     1     1      623            627        -4      933
## 34  2013     1     1      624            630        -6      909
## 35  2013     1     1      624            630        -6      840
## 36  2013     1     1      627            630        -3     1018
## 37  2013     1     1      628            630        -2     1137
## 38  2013     1     1      628            630        -2     1016
## 39  2013     1     1      629            630        -1      824
## 40  2013     1     1      629            630        -1      721
## # ... with 3.367e+05 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## #   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
View(flights) #shows in the dataviewer

#there are other packages etc that allow you to make interactive datatables.
```

ok, so what can we do with this package

1. pick observations, 2.reorder the rows,
2. pick a variable by its name, 4.create new variables, 5.summarise

for all of these functions, The first argument is a data frame. The subsequent arguments describe what to do with the data frame

filter

This lets you pick rows that match an argument -how it works. filter looks for TRUE conditions.

```
filter(flights, month == 1, day == 1) #flights from jan 1st,
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 832 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
filter(flights, month == 5, day == 3) #flights from may 3rd
```

```
## # A tibble: 978 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     5     3       32           2029       243      215
##  2  2013     5     3      152           2159       233      304
##  3  2013     5     3      453            500        -7      657
##  4  2013     5     3      510            515        -5      749
##  5  2013     5     3      538            540        -2      848
##  6  2013     5     3      540            545        -5      815
##  7  2013     5     3      547            600       -13      704
##  8  2013     5     3      550            600       -10      646
##  9  2013     5     3      550            550         0      852
## 10  2013     5     3      552            600        -8      804
## # ... with 968 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
#how would you find all the flights that left at 6am? how many flights left at 6 am from NYC airports
filter(flights, dep_time < 600)
```

```
## # A tibble: 8,730 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 8,720 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
#but, this isn't saved...it isn't changing anything but rather #subsetting data. to save it we need to

may3 <- filter(flights, month == 5, day == 3) #saves results in new tibble
```

```
#remember to use == and not =..also useful is the dplyr::near() function
sqrt(2)^2 == 2
```

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

```
near(sqrt(2)^2, 2)
```

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

```
#ok, but what if we want stuff from both may and june?
#need to use something called boolean
# & = and
# | = or

filter(flights, month == 1 | month ==2) #jan or feb
```

```
## # A tibble: 51,955 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 51,945 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
filter(flights, carrier == "UA" | carrier == "AA")
```

```
## # A tibble: 91,394 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      554            558        -4      740
##  5  2013     1     1      558            600        -2      753
##  6  2013     1     1      558            600        -2      924
##  7  2013     1     1      558            600        -2      923
##  8  2013     1     1      559            600        -1      941
##  9  2013     1     1      559            600        -1      854
## 10  2013     1     1      606            610        -4      858
## # ... with 91,384 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

think about how to ask questions:

lets say we want all flight to Atlanta?

```
filter(flights, dest == "ATL") #but, can't really see the dest col
```

```
## # A tibble: 17,215 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      554            600        -6      812
##  2  2013     1     1      600            600         0      837
##  3  2013     1     1      606            610        -4      837
##  4  2013     1     1      615            615         0      833
##  5  2013     1     1      658            700        -2      944
##  6  2013     1     1      754            759        -5     1039
##  7  2013     1     1      807            810        -3     1043
##  8  2013     1     1      814            810         4     1047
##  9  2013     1     1      830            835        -5     1052
## 10  2013     1     1      855            859        -4     1143
## # ... with 17,205 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

note that we can't see the dest col. . . . there is where the next function comes in 'select'

select

```
select(flights, dest, month, day)
```

```
## # A tibble: 336,776 x 3
##     dest month   day
##    <chr> <int> <int>
##  1   IAH     1     1
##  2   IAH     1     1
##  3   MIA     1     1
##  4   BQN     1     1
##  5   ATL     1     1
##  6   ORD     1     1
##  7   FLL     1     1
##  8   IAD     1     1
##  9   MCO     1     1
## 10   ORD     1     1
## # ... with 336,766 more rows
```

```
#select has lots of options...

select(flights, starts_with("dep"))
```

```
## # A tibble: 336,776 x 2
##    dep_time dep_delay
##       <int>     <dbl>
##  1      517          2
##  2      533          4
##  3      542          2
##  4      544         -1
##  5      554         -6
##  6      554         -4
##  7      555         -5
##  8      557         -3
##  9      557         -3
## 10      558         -2
## # ... with 336,766 more rows
```

```
#try to find out the other options that select has


#say we want flights that flew to Atlanta, and to view the airlines that flew there
# we can chain together things easily to make code easy to write and to read
#chain = 'then'...tells R to do one thing, then do something else...written as "%>%"
#example

flights %>% select(carrier, dest) %>% filter(dest == "ATL")
```

```
## # A tibble: 17,215 x 2
##    carrier  dest
##      <chr> <chr>
##  1      DL   ATL
##  2      MQ   ATL
##  3      DL   ATL
##  4      DL   ATL
##  5      DL   ATL
##  6      DL   ATL
##  7      DL   ATL
##  8      FL   ATL
##  9      MQ   ATL
## 10      DL   ATL
## # ... with 17,205 more rows
```

```
#or
flights %>% select(carrier, arr_delay) %>% filter(arr_delay > 80)
```

```
## # A tibble: 19,556 x 2
##    carrier arr_delay
##      <chr>     <dbl>
##  1      MQ       137
##  2      MQ       851
##  3      UA       123
##  4      UA       145
##  5      MQ        81
##  6      MQ        93
##  7      EV       103
##  8      EV        84
##  9      B6        83
## 10      EV       127
## # ... with 19,546 more rows
```

arrange

```
arrange(flights, year, month)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
arrange(flights, desc(dep_delay)) #sort descending
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     9      641            900      1301     1242
##  2  2013     6    15     1432           1935      1137     1607
##  3  2013     1    10     1121           1635      1126     1239
##  4  2013     9    20     1139           1845      1014     1457
##  5  2013     7    22      845           1600      1005     1044
##  6  2013     4    10     1100           1900       960     1342
##  7  2013     3    17     2321            810       911      135
##  8  2013     6    27      959           1900       899     1236
##  9  2013     7    22     2257            759       898      121
## 10  2013    12     5      756           1700       896     1058
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
arrange(flights, dep_delay)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013    12     7     2040           2123       -43       40
##  2  2013     2     3     2022           2055       -33     2240
##  3  2013    11    10     1408           1440       -32     1549
##  4  2013     1    11     1900           1930       -30     2233
##  5  2013     1    29     1703           1730       -27     1947
##  6  2013     8     9      729            755       -26     1002
##  7  2013    10    23     1907           1932       -25     2143
##  8  2013     3    30     2030           2055       -25     2213
##  9  2013     3     2     1431           1455       -24     1601
## 10  2013     5     5      934            958       -24     1225
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

```
flights %>% select(carrier, arr_delay) %>% arrange(arr_delay)
```

```
## # A tibble: 336,776 x 2
##    carrier arr_delay
##      <chr>     <dbl>
##  1      VX       -86
##  2      VX       -79
##  3      UA       -75
##  4      AA       -75
##  5      AS       -74
##  6      UA       -73
##  7      DL       -71
##  8      UA       -71
##  9      B6       -71
## 10      VX       -70
## # ... with 336,766 more rows
```

```
flights %>% select(carrier, arr_delay) %>% arrange(desc(arr_delay))
```

```
## # A tibble: 336,776 x 2
##    carrier arr_delay
##      <chr>     <dbl>
##  1      HA      1272
##  2      MQ      1127
##  3      MQ      1109
##  4      AA      1007
##  5      MQ       989
##  6      DL       931
##  7      DL       915
##  8      DL       895
##  9      AA       878
## 10      MQ       875
## # ... with 336,766 more rows
```

mutate

adds new stuff notes: it takes a vector of values as the input and returns a new vector…can do almost anything to the data. good way to explore and play with data. i.e. log your data, cumulative sum (cumsum) but it doesn't save it… mutate adds new columns at the end of your dataset so we will start by creating a narrower dataset so we can see the new variables. Remember that when you are in RStudio, the easiest way to see all the columns is View()

```
flights_sml <- select(flights,
                      year:day,
                      ends_with("delay"),
                      distance,
                      air_time) #here, i'm just selecting a few cols
```

```
View(flights_sml)

mutate (flights_sml, hours = air_time/60) #adds col of time in hours
```

```
## # A tibble: 336,776 x 8
##     year month   day dep_delay arr_delay distance air_time     hours
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>     <dbl>
##  1  2013     1     1         2        11     1400      227 3.7833333
##  2  2013     1     1         4        20     1416      227 3.7833333
##  3  2013     1     1         2        33     1089      160 2.6666667
##  4  2013     1     1        -1       -18     1576      183 3.0500000
##  5  2013     1     1        -6       -25      762      116 1.9333333
##  6  2013     1     1        -4        12      719      150 2.5000000
##  7  2013     1     1        -5        19     1065      158 2.6333333
##  8  2013     1     1        -3       -14      229       53 0.8833333
##  9  2013     1     1        -3        -8      944      140 2.3333333
## 10  2013     1     1        -2         8      733      138 2.3000000
## # ... with 336,766 more rows
```

```
#also, transmute makes new data from old

aa <- transmute(flights, hours = air_time / 60)
aa
```

```
## # A tibble: 336,776 x 1
##        hours
##        <dbl>
##  1 3.7833333
##  2 3.7833333
##  3 2.6666667
##  4 3.0500000
##  5 1.9333333
##  6 2.5000000
##  7 2.6333333
##  8 0.8833333
##  9 2.3333333
## 10 2.3000000
## # ... with 336,766 more rows
```

what would this do:

flights %>% select(distance, air_time) %>% mutate(speed = distance/air_time*60) %>% arrange(speed)

summarise and group_by

```
#now, we will talk about summarise and group_by, some of the most impt ones
#lets say we want to know the avg depature delay from the NYC airports
#this is what summarise does, but it is a bit tricky
summarise(flights, delay = mean(dep_delay))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1    NA
```

```
#if you run the above code, you will get a tibble back with "NA." that is because
#not all flights have dep_delay data. we need to explicitly tell R to skip those
#to do that, we use "na.rm = TRUE"
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 x 1
##      delay
##      <dbl>
## 1 12.63907
```

```
# group_by changes the unit of analysis from the complete dataset to individual groups.
#trick: if you are thinking "I want data for each site/airline/population" then group_by is way to go


#if we applied the same code to above to data grouped by date, we get the average delay per date:


flights %>% group_by(dest) %>% summarise(average_delay = mean(dep_delay, na.rm=TRUE))
```

```
## # A tibble: 105 x 2
##     dest average_delay
##    <chr>         <dbl>
##  1   ABQ     13.740157
##  2   ACK      6.456604
##  3   ALB     23.620525
##  4   ANC     12.875000
##  5   ATL     12.509824
##  6   AUS     13.025641
##  7   AVL      8.190114
##  8   BDL     17.720874
##  9   BGR     19.475000
## 10   BHM     29.694853
## # ... with 95 more rows
```

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     year month   day     delay
##    <int> <int> <int>     <dbl>
##  1  2013     1     1 11.548926
##  2  2013     1     2 13.858824
##  3  2013     1     3 10.987832
##  4  2013     1     4  8.951595
##  5  2013     1     5  5.732218
##  6  2013     1     6  7.148014
##  7  2013     1     7  5.417204
##  8  2013     1     8  2.553073
##  9  2013     1     9  2.276477
## 10  2013     1    10  2.844995
## # ... with 355 more rows
```

```
#more on group_by
```

```
by_dest <- group_by(flights, dest)  # group flight by dest. how many destinations are there?

delay <- summarise(by_dest,
                   count = n(),
                   dist = mean(distance, na.rm = TRUE),
                   delay = mean(arr_delay, na.rm = TRUE))   # use summerize to compute distance, avg de

delay %>%  group_by(dest) %>% nest()
```

```
## # A tibble: 105 x 2
##     dest             data
##    <chr>           <list>
##  1   ABQ <tibble [1 x 3]>
##  2   ACK <tibble [1 x 3]>
##  3   ALB <tibble [1 x 3]>
##  4   ANC <tibble [1 x 3]>
##  5   ATL <tibble [1 x 3]>
##  6   AUS <tibble [1 x 3]>
##  7   AVL <tibble [1 x 3]>
##  8   BDL <tibble [1 x 3]>
##  9   BGR <tibble [1 x 3]>
## 10   BHM <tibble [1 x 3]>
## # ... with 95 more rows
```

```
delay <- filter(delay, count > 20, dest != "HNL") #not HNL
```

Tidy data

What is tidy data?

it is a way of having the data set so that R can work on it well. the ‘oddest’ part about tidy data is the way it is set up is often counter to how we normally think of Excel etc

example is this data from our current project >TempUSUkUSSR.csv (I hopefully will remember to send this along with this file. if not please remind me)

for data to be Tidy 1. each variable gets its own column 2. each observation has its own row 3. each value has its own cell

- from my experience, 1&2 are the ones that we need to work on

lets look at this data

```
library(tidyverse)
My_data <- read_csv("data/TempUSUkUSSR.csv")
```

```
## Parsed with column specification:
## cols(
##   Year = col_integer(),
##   Temp_Diff = col_double(),
##   USA = col_double(),
##   UK = col_double(),
##   Russia = col_double()
## )
```

```
glimpse(My_data)
```

```
## Observations: 133
## Variables: 5
## $ Year      <int> 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888...
## $ Temp_Diff <dbl> -0.40, -0.35, -0.32, -0.39, -0.59, -0.60, -0.53, -0....
## $ USA       <dbl> -0.17092753, -0.17195357, -0.17295087, -0.17541935, ...
## $ UK        <dbl> -0.016270896, -0.016940181, -0.017318444, -0.0195490...
## $ Russia    <dbl> -0.04010697, -0.04126685, -0.04238522, -0.05008476, ...
```

Why is this not Tidy? well, the USA/UK/Russia cols are not variables, but values of a variable

how to make it Tidy? - first, we need a new column with a variable name. lets call this 'country'. this new variable name is called the *Key* - then, we need to know the name of the cases. in this example, those values are the scaled DCI. I'm going to call it DCI_scaled. this is called the *value*

-then, we can let R do magic

```
My_data2 <- My_data %>% gather(USA:Russia, key = country, value = DCI)

glimpse(My_data2)
```

```
## Observations: 399
## Variables: 4
## $ Year      <int> 1880, 1881, 1882, 1883, 1884, 1885, 1886, 1887, 1888...
## $ Temp_Diff <dbl> -0.40, -0.35, -0.32, -0.39, -0.59, -0.60, -0.53, -0....
## $ country   <chr> "USA", "USA", "USA", "USA", "USA", "USA", "USA", "US...
## $ DCI       <dbl> -0.17092753, -0.17195357, -0.17295087, -0.17541935, ...
```

Hooray!!!!

Other things to keep in mind when wrangling code Separate is a useful one it takes a col and makes it into multiple cols Note: you can do a lot of this with Excel. but the point is that this makes every step you made crystal clear!

so now we can get mean by country

```
My_data2 <- My_data %>% gather(USA:Russia, key = country, value = DCI)
My_data2 %>% group_by(country) %>% summarise(mean = mean(DCI, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   country         mean
##     <chr>        <dbl>
## 1  Russia -0.026589814
## 2      UK -0.003162897
## 3     USA -0.063837662
```

Visualzation

For this section we are going to use the gapminder package, which i load below

```
library(gapminder)
```

```
## Warning: package 'gapminder' was built under R version 3.4.4
```

there are numerous plotting packages. Ggplot2 has become the standard for a number of reasons. it is based on an idea called the Grammar of Graphics by Leland Wilkinson. this package was developed by Hadley Wickham. his book is pretty useful here https://www.amazon.com/ggplot2-Elegant-Graphics-Data-Analysis/dp/331924275X/ref=as_li_ss_tl?ie=UTF8&linkCode=sl1&tag=ggplot2-20&linkId=4b4de5146fdafd09b8035e8aa656f300

From the Wickham book we learn that Every ggplot2 plot has three key components:

1. data,
2. A set of aesthetic mappings between variables in the data and visual properties
3. At least one layer which describes how to render each observation. Layers are usually created with a geom function.

Another great resource is this book http://socviz.co/index.html#preface I learned a lot from it and as of now it is free online

a lot of the below comes from that book

in ggplot, the connections between your data and the plot elements are called *aesthetic mappings* or just *aesthetics*.

1. we begin every plot by telling the ggplot() function what your data is, and how the variables map onto the aesthetics.

2. Then we tell it what kind of plot (called a **geom**) we want. can be a scatterplot, a histogram, etc. geom_point() makes scatterplots and geom_boxplot() makes boxplots

3. we then combine these two pieces, the ggplot() object and the geom, by adding them together in an expression, using the + symbol.

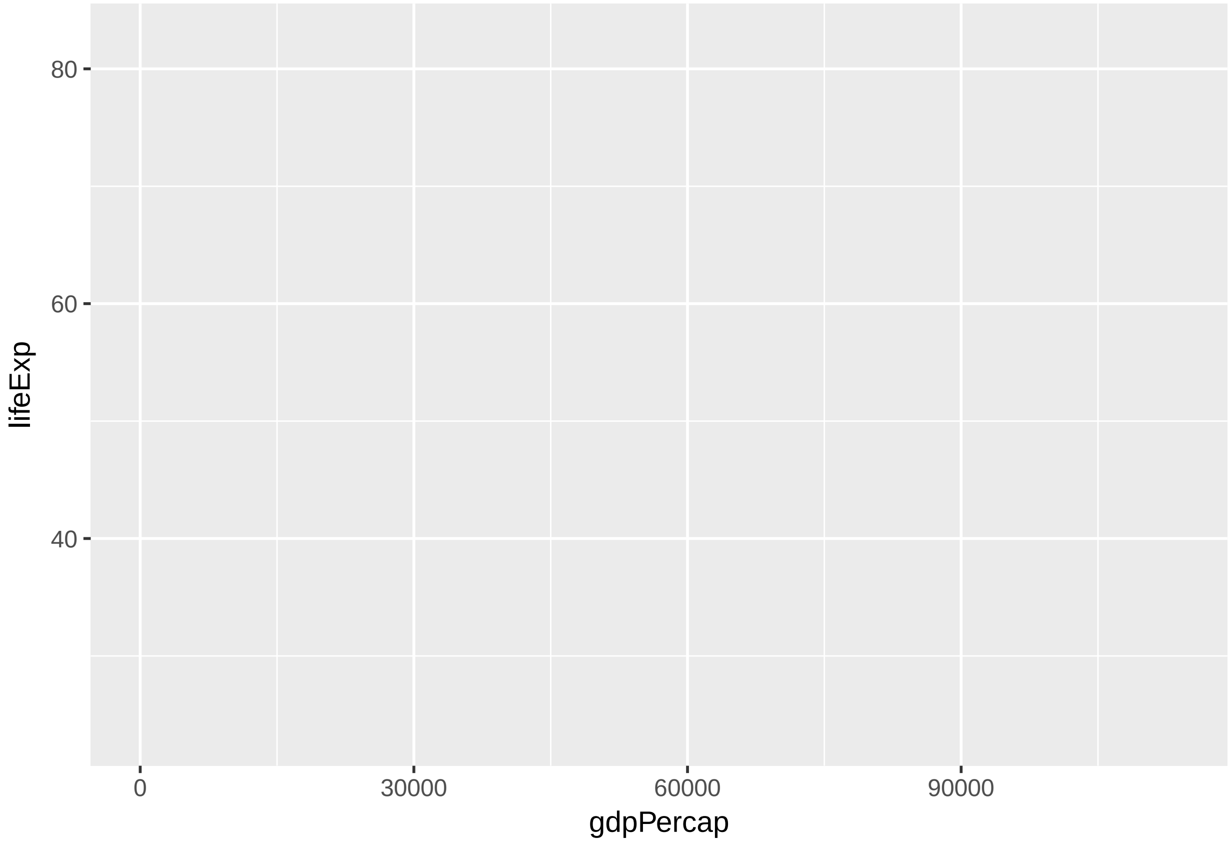
general steps: steps: " 1.Tell the ggplot() function what our data is. The data = step. 2.Tell ggplot() what relationships we want to see.The mapping = aes step. For convenience we will put the results of the first two steps in an object called p. 3.Tell ggplot how we want to see the relationships in our data. Choose a geom. 4. Layer on geoms as needed, by adding them to the p object one at a time. 5. Use The scale_, family, labs() and guides() functions. some additional functions to adjust scales, labels, tick marks, titles. "

```r
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp))
```

the above code says "make a r object called p. this is going to use the data in the gapminder data table and we are going to want the x-axis to be gdpPercap and the y-axis to be lifeExp"

if you then told R to print p you'd see an empty plot

```r
p
```

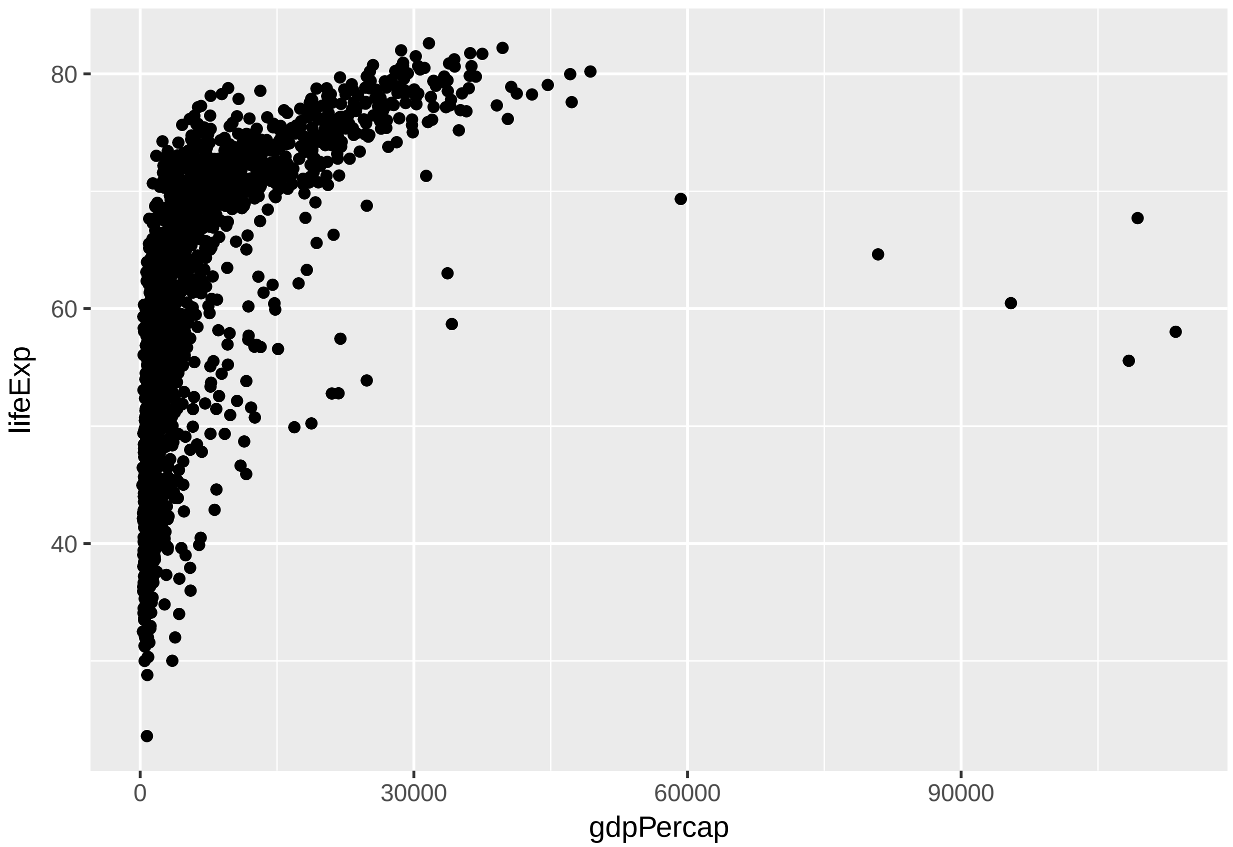


This is a happy thing! it has all the data there, but it doesn't know what we want to do. if you were to check the structure of p (can do so with str(p)) you would see all kinds of thing behind the scenes.

the next step is to add layers to plots

when adding layers we dont need to tell ggplot whee the data is coming from since it inherits it from the original object. but we can change where the data is coming from if we wish!

```
p + geom_point()
```

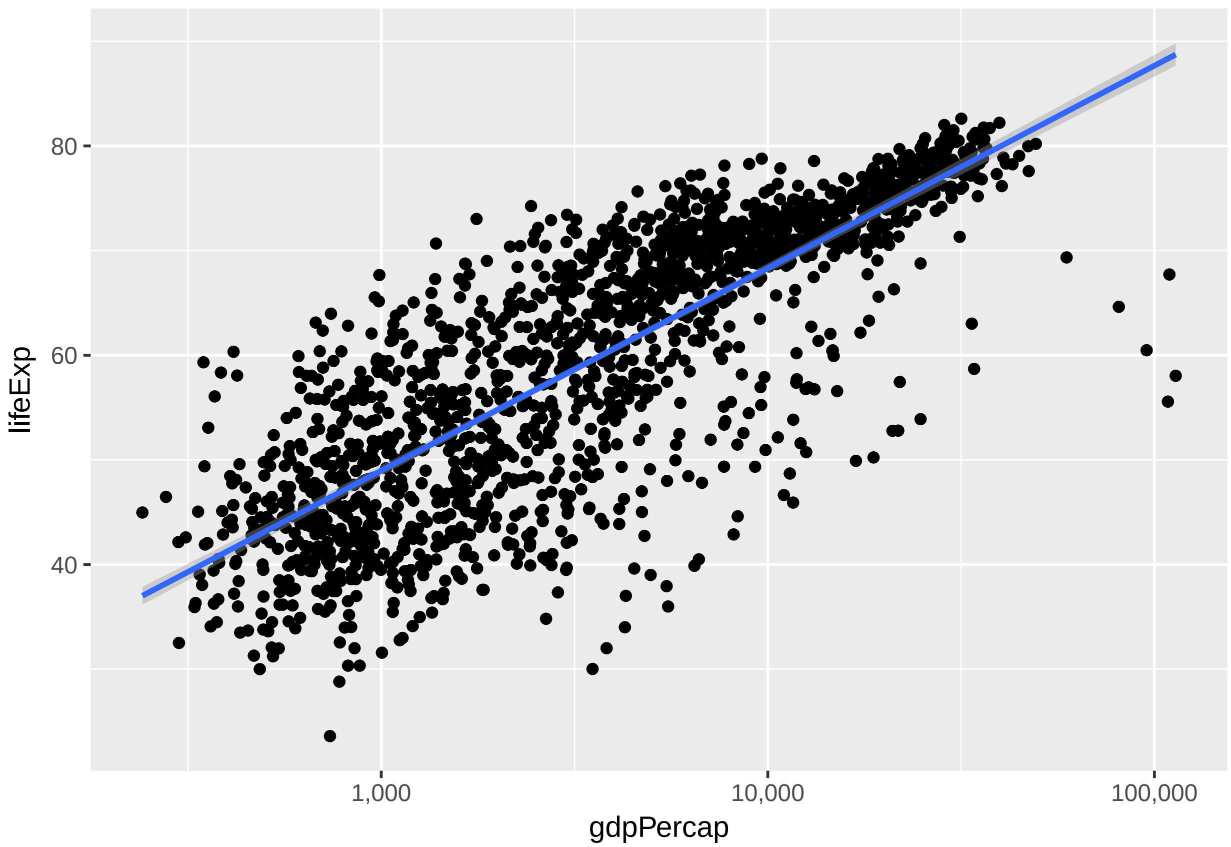


SCALES

can transform the x and y axis directly with scale_x_log10 etc. this will transform axis and tick marks will be in scientific notions. we can also give scale_X_log a label that reformat the text under the axis

protip: look at the line "scale_x_log10(labels =scales::comma)" the labels = scales::comma is kinda odd. what is going on here is that we are calling a function, in this case 'comma', from the scales package, without loading that package. this is sometimes useful and a good move to keep in your back pocket

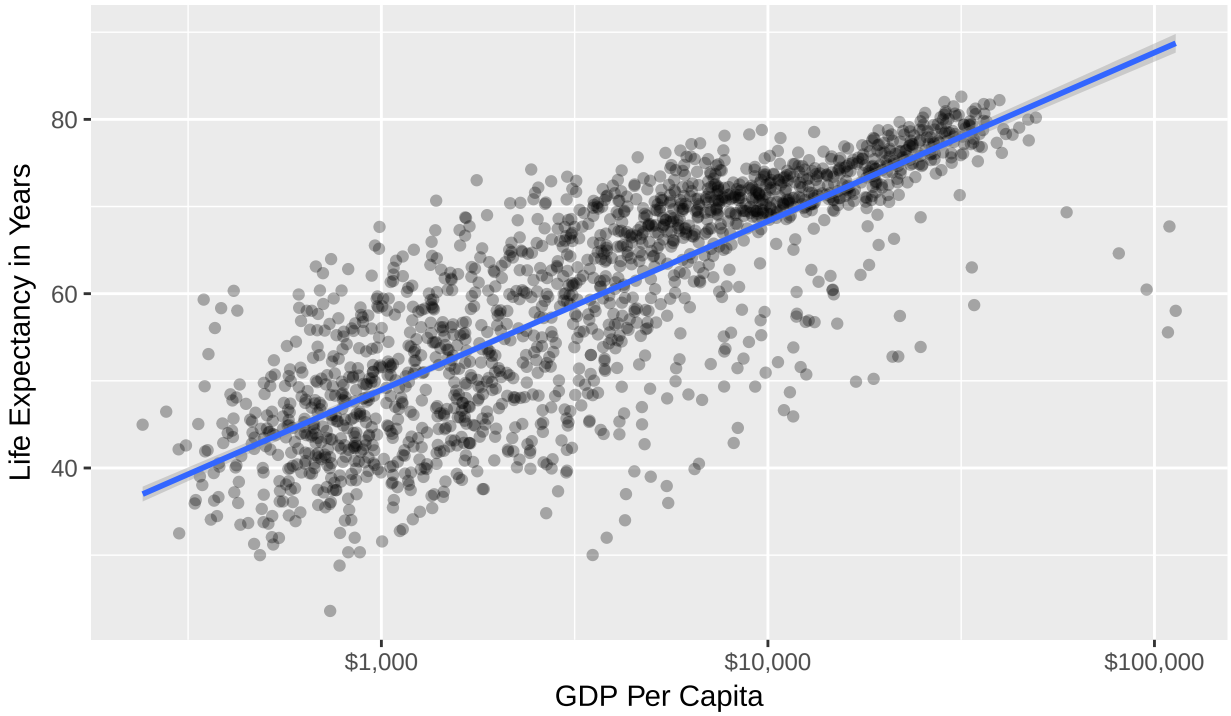
```
p <- ggplot(data = gapminder, mapping = aes(x = gdpPercap, y=lifeExp))
p + geom_point() + geom_smooth(method = "gam") + scale_x_log10(labels =scales::comma)
```



```r
p <-  ggplot(data = gapminder,
             mapping = aes(x = gdpPercap,
                           y = lifeExp,
                           color = continent))
```

Adding lables to the graph

```r
p <- ggplot(data = gapminder, mapping = aes(x = gdpPercap, y=lifeExp))
p + geom_point(alpha = 0.3) +
    geom_smooth(method = "gam") +
    scale_x_log10(labels = scales::dollar) +
    labs(x = "GDP Per Capita", y = "Life Expectancy in Years",
         title = "Economic Growth and Life Expectancy",
         subtitle = "Data points are country-years",
         caption = "Source: Gapminder.")
```


Economic Growth and Life Expectancy

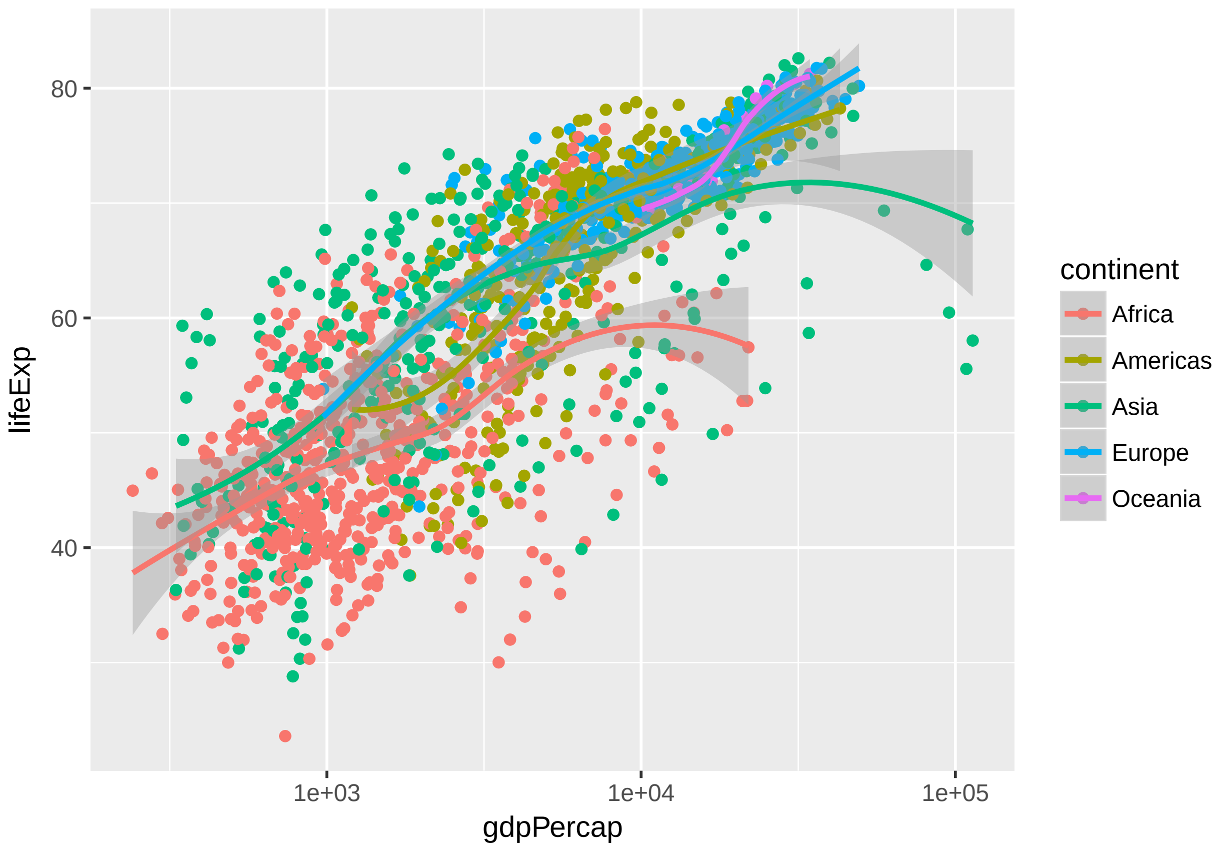
Data points are country−years

Source: Gapminder.

Adding groups

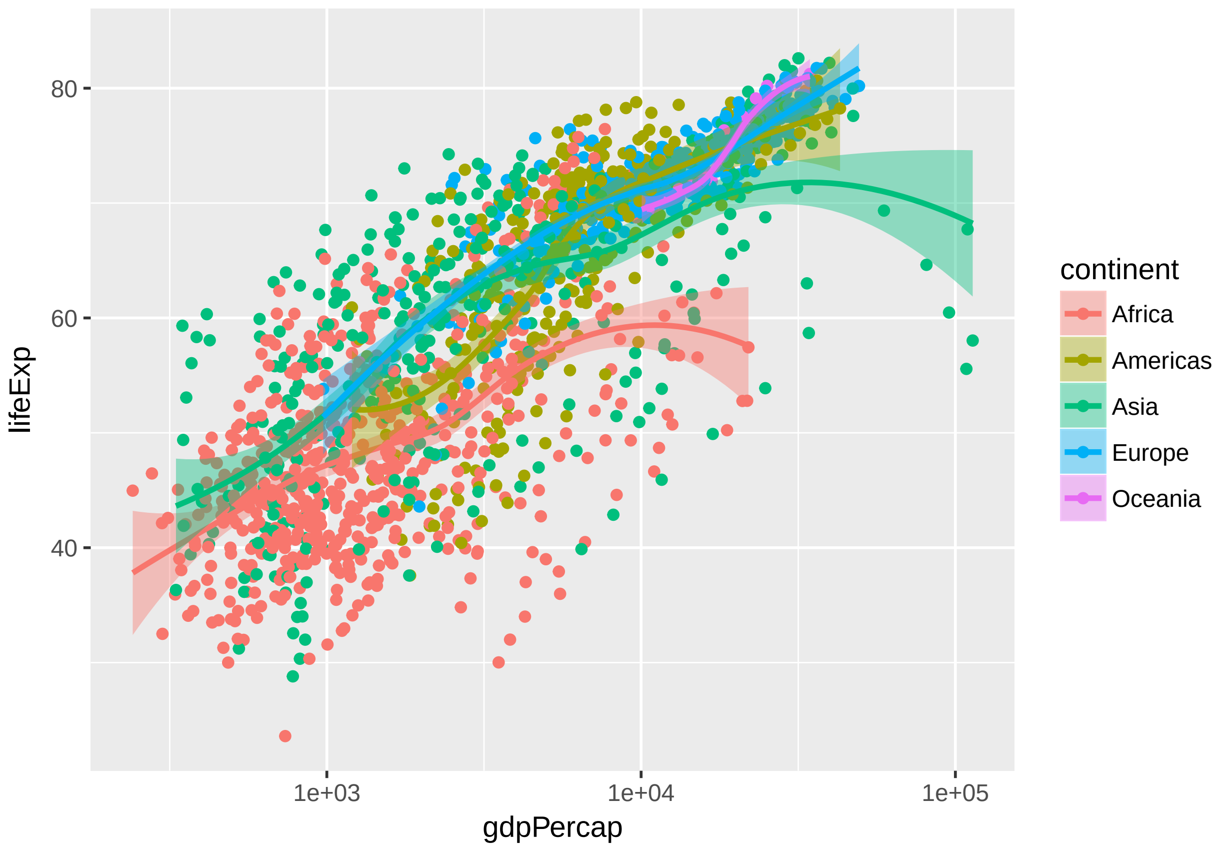
When setting aesthetic we can also make a new mapping. often we want to group things together. can group by color, shape, and size. fun to play with these things

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp,
                          color = continent))
p + geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10()
```



let's say we want the color of the standard error ribbon to match that of the dominant color. this color is controlled by the fill aesthetic like this

```
p <- ggplot(data = gapminder,
            mapping = aes(x = gdpPercap,
                          y = lifeExp,
                          color = continent,
                          fill = continent))
p + geom_point() +
    geom_smooth(method = "loess") +
    scale_x_log10()
```



As you play around with ggplot you can learn all kinds of tricks and tips!