EDA_Tidy_Notes

Marc Kissel

2/2/2019

today we are going to start with how to explore data.

1. first part of the day i will do a quick demo of how someone might apporach a dataset.
2. then, will talk more on data transformation 2.5 tidy data
3. then group projects
4. then maybe more plots

Pro-tip: good solutions and good code comes from starting with bad code and working though it!

Live data analysis via TidyTuesday

2. Data transformation

ok, so how do we take our data and explore it!

Data transformation is the key to R. we are going to take data in from a worksheet/csv/whatever and transform it.

There are a number of ways to do this. for me, the method that gives us the most options is using a package called *dplyr.* This is part of a larger group of packages known as the *Tidyverse.* we are also going to use a packaged called nycflights13

```
#install.packages("nycflights13")
library(tidyverse)
```

```
## -- Attaching packages --------------- tidyverse 1.2.1 --

## v ggplot2 3.0.0     v purrr   0.2.5
## v tibble  1.4.2     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.1.1     v forcats 0.3.0

## Warning: package 'ggplot2' was built under R version 3.5.1

## Warning: package 'tidyr' was built under R version 3.5.2

## Warning: package 'dplyr' was built under R version 3.5.2

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

```
library(nycflights13)
```

```
## Warning: package 'nycflights13' was built under R version 3.5.1
```

```
#you could load  the dplyr library by itself
```

Lets start by looking at the data we want to explore. to do this we can simply write the name of the object

```r
flights
```

```
## # 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>
```

Question 1

a. how many observations are there in the flights dataset?

b. how many columns?

If you want to just look at the data quickly, the *head* and *print* function works well

```r
head(flights)
```

```
## # A tibble: 6 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
## # ... with 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>
```

```r
print(flights)
```

```
## # 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>
```

question 2

a) how many rows does the print function give by default? how would you update the function call to make it give you the first 32 rows?

Finally, we can use the built-in dataviewer:

```
View(flights) #note that View is  capatilzed
```

ok, so what can we do with dplyr?

pick observations, reorder the rows, pick a variable by its name, create new variables, and summarise

###this is not that easy. takes some practice to get used to the different **verbs**

all verbs work the same 1. first argument is a dataframe 2. the next ones talk about what to do to the dataframe

Filter

filter selects rows based on an arguement. it looks for when somethign is TRUE

```
#lets say i want the first of jan flights
filter(flights, month ==1, day == 1)
```

```
## # 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>
```

Question

How would you find all the flights that left at 6:00am? how many flights were dealyed more than 30 mins

but, this isn't saved…it isn't changing anything but rather subsetting data. to save we need to make new dataframe

```r
may3 <- filter(flights, month == 5, day == 3)
```

also useful is a fun fuctnion called near, which has built in tolerance to variation

```r
sqrt(2) ^ 2 == 2
```

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

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

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

ok, but what if we want something or something else

##boolean

& = and | = or

```r
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>
```

Question

How would you find all the flights that left at 6:00am? how many flights were dealyed more than 30 mins : Find all flights that flew to Atlanta

#Select this verb keeps only the vars we name

```r
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
```

fun!

Question

a) Select has a lot of options. take a look at the help. how would you

- select only the vars that start with prefix 'dep'

say we want flights that flew to Atlanta, and to view the airlines that flew there

A good way is to *chain* arguments together using a *pipe* : %>% This pipe is read as "then". so do one thing and then do another

```r
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
```

```r
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
```

#arrange

Like with many spreadsheets we can sort the rows with this verb

```r
arrange(flights, 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))
```

```
## # 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>
```

#mutate

add new stuff with mutate 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, cumulate sum (cumsum)

```
flights_sml <- select(flights,
                      year:day,
                      ends_with("delay"),
                      distance,
                      air_time)
```

what did the above code do?

```
View(flights_sml)
mutate (flights_sml, hours = air_time/60)
```

```
## Warning: The `printer` argument is soft-deprecated as of rlang 0.3.0.
## This warning is displayed once per session.
```

```
## # 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 3.78
##  2  2013     1     1         4        20     1416      227 3.78
##  3  2013     1     1         2        33     1089      160 2.67
##  4  2013     1     1        -1       -18     1576      183 3.05
##  5  2013     1     1        -6       -25      762      116 1.93
##  6  2013     1     1        -4        12      719      150 2.5
##  7  2013     1     1        -5        19     1065      158 2.63
##  8  2013     1     1        -3       -14      229       53 0.883
##  9  2013     1     1        -3        -8      944      140 2.33
## 10  2013     1     1        -2         8      733      138 2.3
## # ... with 336,766 more rows
```

What does the code below do? try to figure out before you run it to see how

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

#summarise llets say we want to know the avg depature delay from the NYC airports

this can get tricky due to the defaults

```
summarise(flights, delay = mean(dep_delay))
```

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

hmmm...that isn't right...

ah, the mean function keeps nas by default

```
summarise(flights, delay = mean(dep_delay, na.rm = T))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1  12.6
```

#group_by

this verb 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

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


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 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
```

```
## 6  2013     1     6  7.15
## 7  2013     1     7  5.42
## 8  2013     1     8  2.55
## 9  2013     1     9  2.28
## 10 2013     1    10  2.84
## # ... with 355 more rows
```

```
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

#question: what if i want only places that had more than 20 flithts to an airport
delay <- filter(delay, count > 20, dest != "HNL") #not HNL
```

questions

ok, so how would you figure out all the flights that are going to Atlanta, grouped by airline, and seeing the avg. time it take to get there

next, find all flights where the carrier is DL and then make a new column that lists the sched_arr_time in hours rather than minutes then take this new data set and save it to a new variable.

there is another paackae in tidyverse called readr. view the help and figure out how to export your new dataset!

next section: 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 spreadsheets

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

##first, a quick example to show you how it works in *real world*

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

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

```r
View(My_data)
```

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

#show image

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

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

```
## # A tibble: 399 x 4
##     Year Temp_Diff country    DCI
##    <int>     <dbl> <chr>    <dbl>
##  1  1880     -0.4  USA     -0.171
##  2  1881     -0.35 USA     -0.172
##  3  1882     -0.32 USA     -0.173
##  4  1883     -0.39 USA     -0.175
##  5  1884     -0.59 USA     -0.173
##  6  1885     -0.6  USA     -0.174
##  7  1886     -0.53 USA     -0.175
##  8  1887     -0.47 USA     -0.176
##  9  1888     -0.42 USA     -0.177
## 10  1889     -0.27 USA     -0.178
## # ... with 389 more rows
```

#as a reminder, we can now dplyr this

lets divide the year into mil, cent, and year

```r
My_data %>% separate(Year,  into =c("mi", "century", "year"), sep = c(1, 2))
```

```
## # A tibble: 133 x 7
##    mi    century year  Temp_Diff    USA      UK  Russia
##    <chr> <chr>   <chr>     <dbl>  <dbl>   <dbl>   <dbl>
##  1 1     8       80        -0.4  -0.171 -0.0163 -0.0401
##  2 1     8       81        -0.35 -0.172 -0.0169 -0.0413
##  3 1     8       82        -0.32 -0.173 -0.0173 -0.0424
##  4 1     8       83        -0.39 -0.175 -0.0195 -0.0501
##  5 1     8       84        -0.59 -0.173 -0.0197 -0.0510
##  6 1     8       85        -0.6  -0.174 -0.0164 -0.0519
##  7 1     8       86        -0.53 -0.175 -0.0148 -0.0516
##  8 1     8       87        -0.47 -0.176 -0.0135 -0.0514
##  9 1     8       88        -0.42 -0.177 -0.0138 -0.0484
## 10 1     8       89        -0.27 -0.178 -0.0138 -0.0525
## # ... with 123 more rows
```

#ok, but lets use a easy exaple we can walk through From: http://r4ds.had.co.nz/tidy-data.html

https://r4ds.had.co.nz/tidy-data.html#fig:tidy-structure

why?

There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.

There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. As you learned in mutate and summary functions, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural

Tidy Steps: 1. figure out what the variables and observations are 2. resolve one of two common problems: One *variable* might be spread across multiple columns. One *observation* might be scattered across multiple rows.

to fix this we can use *gather* and *spread*

GAthering

lets say the col names are not names of varibales, but the values of a varibabel: like with the dci example.

#> country `1999` `2000` #> * #> 1 Afghanistan 745 2666 #> 2 Brazil 37737 80488 #> 3 China 212258 213766

The set of columns that represent values, not variables. In this example, those are the columns 1999 and 2000.

The name of the variable whose values form the column names. I call that the key, and here it is year.

The name of the variable whose values are spread over the cells. I call that value, and here it's the number of cases.

gather(`1999`, `2000`, key = "year", value = "cases")

https://d33wubrfki0l68.cloudfront.net/3aea19108d39606bbe49981acda07696c0c7fcd8/2de65/images/tidy-9.png

##spreading

this is kinda the oppoise of gather

when an observation is scattered across multiple rows. For example, take table2: an observation is a country in a year, but each observation is spread across two rows.

#> # A tibble: 12 x 4 #> country year type count #> #> 1 Afghanistan 1999 cases 745 #> 2 Afghanistan 1999 population 19987071 #> 3 Afghanistan 2000 cases 2666 #> 4 Afghanistan 2000 population 20595360 #> 5 Brazil 1999 cases 37737 #> 6 Brazil 1999 population 172006362 #> # … with 6 more rows

table2 %>% spread(key = type, value = count)

#> # A tibble: 6 x 4 #> country year cases population #> #> 1 Afghanistan 1999 745 19987071 #> 2 Afghanistan 2000 2666 20595360 #> 3 Brazil 1999 37737 172006362 #> 4 Brazil 2000 80488 174504898 #> 5 China 1999 212258 1272915272 #> 6 China 2000 213766 1280428583

https://d33wubrfki0l68.cloudfront.net/8350f0dda414629b9d6c354f87acf5c5f722be43/bcb84/images/tidy-8.png

more examples:

mini_iris <- iris[c(1, 51, 101),] mini_iris %>% gather(key="flower_measurments", value = "metric")

gather(mini_iris, key = "flower_att", value = "measurement", Sepal.Length, Sepal.Width, Petal.Length, Petal.Width)

mini_iris %>% gather(key="flower_measurments", value = "metric", -Species)

###anthoer example

this time lets build a data set

```
non_tidy <- data.frame(
  box = c("one", "two", "three"),
  male = c(56,78,90),
  female = c(59,34,23)
```

```
)
```

non_tidy %>% gather(key = “sex”, value = ‘number’, male, female)

messy <- data.frame(name = c(“Wilbur”, “Petunia”, “Gregory”), a = c(67, 80, 64), b = c(56, 90, 50)) m

#live example

3. EDA

---

much of this comes from examples from R4DS

##What is EDA? - idea is to generate questions about the data – such as what types of variation do i see – how do the varibles covary - search for answers (by visualizing, transforming, and modeling) - then, use what we learn to refine our questions

first step is often looking a the data. we will do this more in a few weeks but for now some basics

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

one part of the **Tidyverse** is a package called ggplot2. we are going to use this to explore the diamonds dataset.

First, lets look at the info that comes with the dataset

```
?diamonds
```

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

take a minute and read the help. what info is here?

Now we want to take a look at the actual dataset. there are a few ways we can do this, each with their own pros/cons

1. str

```
str(diamonds)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
```

```
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

2. glimpse

```r
glimpse(diamonds)
```

```
## Observations: 53,940
## Variables: 10
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, ...
## $ cut     <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very G...
## $ color   <ord> E, E, E, I, J, J, I, H, E, H, J, J, F, J, E, E, I, J, ...
## $ clarity <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI...
## $ depth   <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, ...
## $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54...
## $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339,...
## $ x       <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, ...
## $ y       <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, ...
## $ z       <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, ...
```

3. print

```r
print(diamonds)
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7 0.24  Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8 0.26  Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9 0.22  Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10 0.23  Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

4. head

```r
head(diamonds) #note default is n = 10
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1 0.23  Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2 0.21  Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3 0.23  Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4 0.290 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5 0.31  Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6 0.24  Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

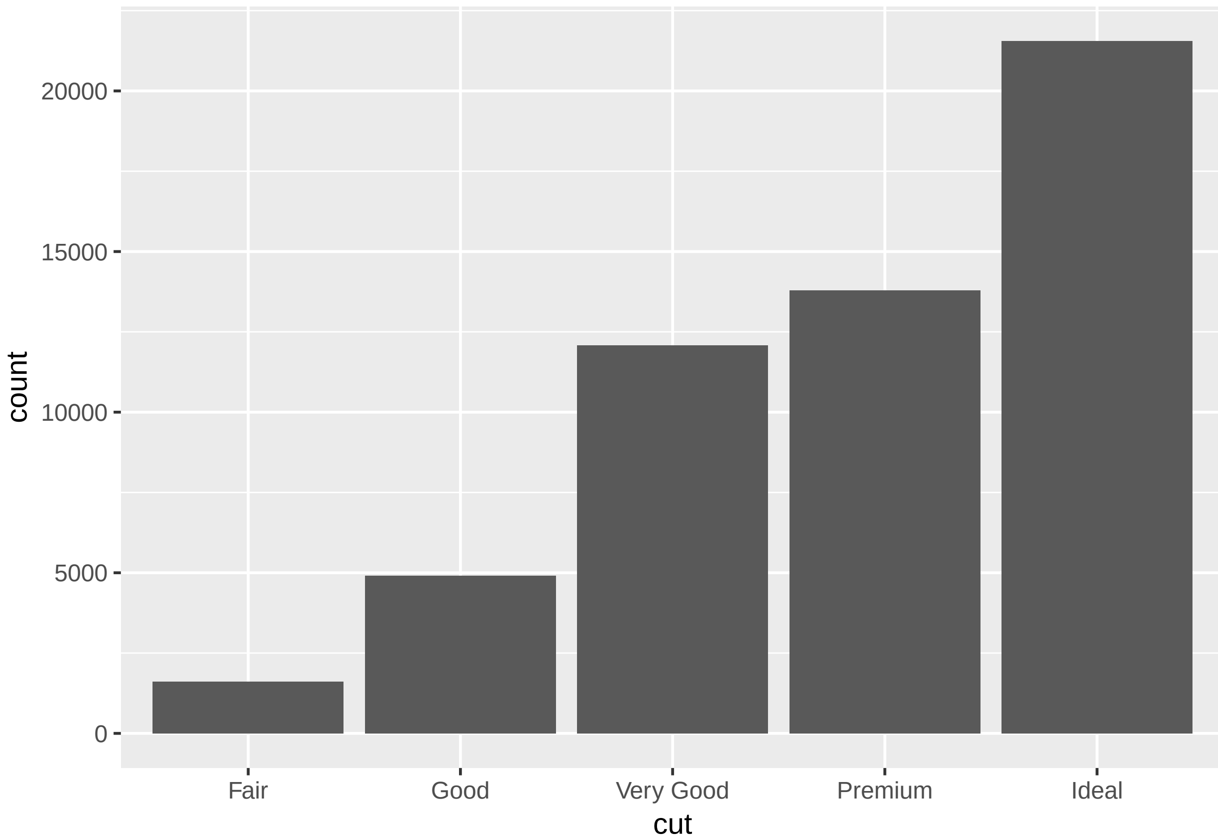
Question 1:

What are the differences between the different ways to see the data

ok, so lets start by making a plot

here is some code

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut))
```



how does this code work? it calls a function called ggplot and tells it that the data we want to use is in the diamonds dataset. we then use a + sign to add a *geom* onto this plot. for now, we can think of a geom as the grammar of the type of plot we want. here i want a bar chart since im looking at categorical data. i set the geom_bar function and tell it to work on the object that has the X-axis as 'cut'. Because we already told R that the data was in diamonds it knows where to look.

congrats! you just made your first plot