Data manipulation + Visualization - Part II

William Ou

6/6/2020

Contents

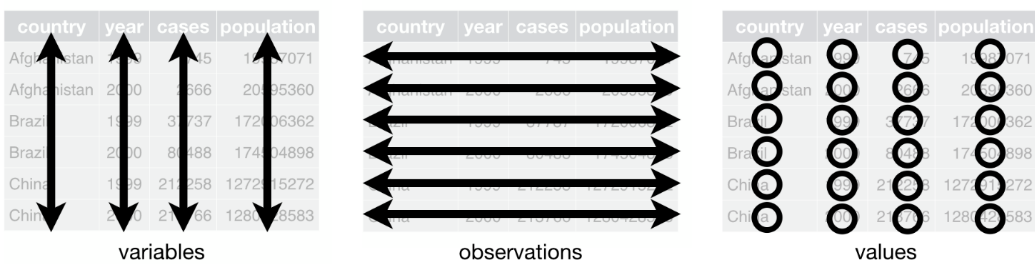
1. Review: i) *tidy* philosophy, ii) indexing
2. Reshaping (pivotting)
3. Data manipulation assignment
4. Visualizing (if time permitting)

1. Review:

i) *tidy* philosophy

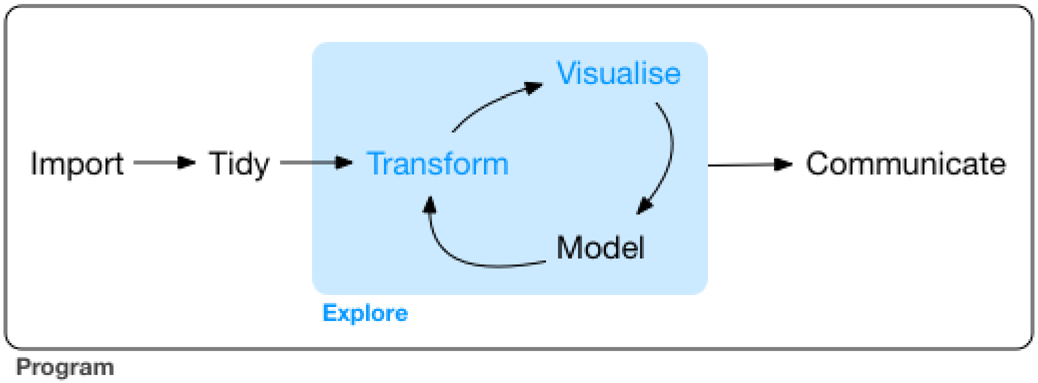
Rules of tidy data

- Each *variable* must have its own *column*
- Each *observation* must have its own *row*
- Each *value* must have its own *cell*



The program:

- Transform data to help highlight/visualize the relationships between different variables
- The more complex the data the more ways you can transform it
- ITERATE!



Tidy code:

- The piping operator *%>%* can help you organize your script such that every line corresponds to just 1 specific action

ii) Indexing

- **INTEGERS** - indexing can be done by supplying a vector of ***integers*** which correspond to the observations in your vector/dataframe/matrix/etc...

```
random_vector <- c(4,19,2,28,115,1832,18)

random_vector[4] #returns the 4th element of the vector
```

```
## [1] 28
```

```
random_vector[c(3,5,1)] #returns the elements 3,5,1 (and in that sequence)
```

```
## [1]   2 115   4
```

- **BOOLEAN** - more commonly, indexing is done by specifying some ***conditional statement*** which in essense returns a vector of type ***boolean*** that is used to subset/index your data

```
random_vector>3 #this gives a vector of TRUE/FALSE in which the TRUE/FALSE correspond to
```

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

```
# the sequence of values in the original vector that satisfies the condition >3

random_vector[random_vector>3] #returns those values that are >3
```

```
## [1]    4   19   28  115 1832   18
```

- Indexing/subsetting by boolean is straightforward when you consider ***just 1 conditional statement*** and more so when that conditional statement returns a vector of TRUE/FALSE of the ***same length*** as the vector you indexing

```
length(random_vector)==length(random_vector>3)
```

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

- Sometimes your code runs but there's actually a bug that you can't see right away

```
random_vector[c(TRUE,FALSE,FALSE)]
```

```
## [1]  4 28 18
```

- the boolean vector only has length 3 but it still manages to subset more than 3 values, what happens is that R will recycle those TRUE/FALSE until iterating through the whole length of the sequence

In other words, supplying c(TRUE,FALSE) will give you every other value

```
random_vector[c(TRUE,FALSE)]
```

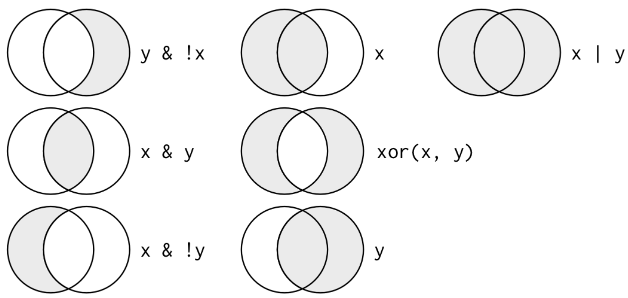
```
## [1]   4   2 115  18
```

- The problem of the example below is more obvious because T/F is longer than the actual vector, leading to production of NAs

```
random_vector[c(TRUE,FALSE,FALSE,TRUE,TRUE,FALSE,FALSE,TRUE,TRUE,TRUE,TRUE)]
```

```
## [1]   4  28 115  NA  NA  NA  NA
```

- When working with dataframes (ie. multiple vectors (columns) that are related to one another (row, observation)), we might want to subset by multiple conditional statements
- ***logical*** operators (ie. if, or, &, etc..) are used to combine and relate boolean vectors:



___An example with with mtcars data:___

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
#select cars with 6 cylinders (condition 1) AND hp less than 110 (condition 2)
mtcars$cyl == 6 #Condition 1
```

```
##  [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
mtcars$hp < 110 #Condition 2
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
## [25] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
```

```
#Index/subset by using & operator
mtcars[mtcars$cyl==6 & mtcars$hp<110,]
```

```
##          mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Valiant 18.1   6  225 105 2.76 3.46 20.22  1  0    3    1
```

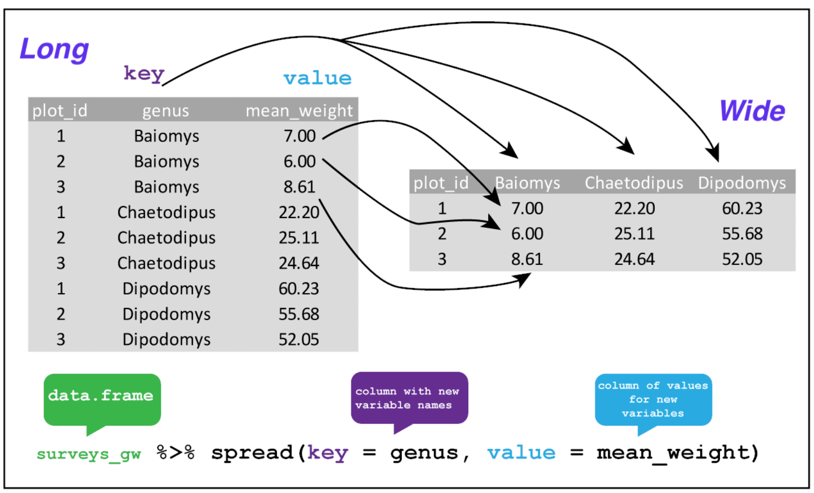
```r
#tidyverse solution
mtcars%>%
  filter(cyl==6 & hp<110) #Note: rownames are removed when using tidyverse functions
```

```
##    mpg cyl disp  hp drat   wt  qsec vs am gear carb
## 1 18.1   6  225 105 2.76 3.46 20.22  1  0    3    1
```

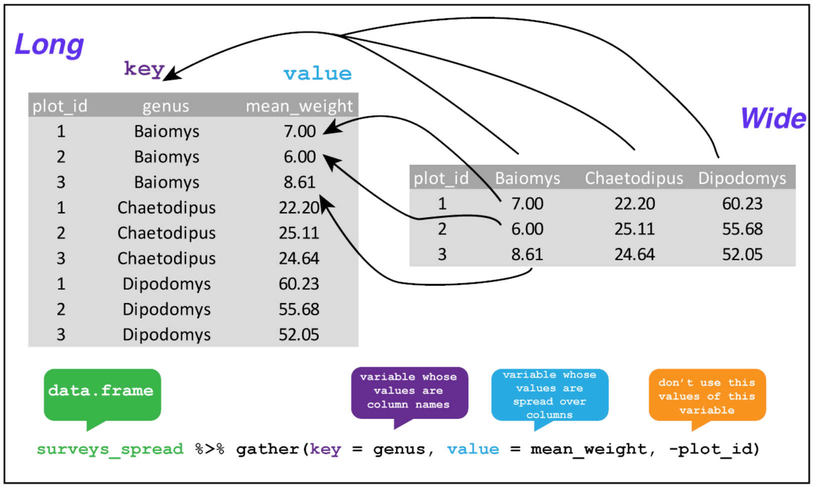
2. Reshaping: transposing/pivoting

- From the *tidy philosophy* section, we talked about the iterative process of data exploration (transform –> visualize –> transform. . .)
- From the *tidy data* section we see that each row should always represent an observation of **n**-variables
- Data transformations therefore provide a way to alter *relationship* between variables
- In tidyverse, reshaping data is done by pivoting (w/ pivot_wider() or pivot_longer())
- NOTE: originally the functions were spread() and gather(), but the idea is the same

Pivot wider (spread)



Pivot longer (gather)



For example, in the zooplankton dataset, I might be interested in the correlation between the abundance of different species. In this case, each row (ie. observation) should represent a unique sampling event in which each species is its own variable with values = to their observed abundances.

To do so: 1. I use group_by() and summarise() to get the counts (ie. abundance) of each species for each lake.

```
# i. Use summary to get counts/abundance of each species for each lake
abundance_data <- clean_data%>%
  group_by(lake,sample,genus)%>%
  summarise(abundance=n())  # the function n() simply counts the frequency of a
#unique genus within each lake within each sample which is equivalent to its abundance

head(abundance_data)
```

```
## # A tibble: 6 x 4
## # Groups:   lake, sample [3]
##   lake     sample genus        abundance
##   <fct>    <fct>  <fct>            <int>
## 1 Green    ZP17   ceriodaphnia        13
## 2 Green    ZP17   cyclopoida         111
## 3 Green    ZP17   rotifer              3
## 4 Lillooet ZP31   calanoid            15
## 5 Lillooet ZP31   cyclopoida         115
## 6 Lillooet ZP32   calanoid            63
```

2. I then use to pivot_wider() function to “spread” the *genus* column into its own individual columns with the *abundance* as its values.

```
species_matrix <- abundance_data%>%
  pivot_wider(names_from = 'genus',values_from = 'abundance')%>%
  replace(is.na(.),0)
#Since the rows in the original data represent only presences (ie. absences do not
# have observations), NAs are produced when transformed to the wide form. The arugment
# values_fill provides a way to fill those NAs
```

- The difference between the original (abundance_data) and new (species_matrix) is that the original focuses on individual zooplanktons (each observation represents an individual) while the new one focuses on the sample (or the community of zooplanktons)

```
nrow(clean_data) #individual zooplankton
```

```
## [1] 864
```

```
nrow(abundance_data) #population-level (genus)
```

```
## [1] 31
```

```
nrow(species_matrix) #community-level (sample)
```

```
## [1] 7
```

- I can now easily get the correlation matrix of species abundances

```
species_matrix%>%ungroup%>%select(-lake,-sample)%>%cor
```

```
##              ceriodaphnia  cyclopoida     rotifer    calanoid     bosmina
## ceriodaphnia   1.00000000  0.34462971  0.06275655 -0.48758686 -0.26932490
## cyclopoida     0.34462971  1.00000000 -0.06278958 -0.16403043 -0.73732820
## rotifer        0.06275655 -0.06278958  1.00000000 -0.40933958 -0.06011326
## calanoid      -0.48758686 -0.16403043 -0.40933958  1.00000000 -0.25169330
## bosmina       -0.26932490 -0.73732820 -0.06011326 -0.25169330  1.00000000
## copepod        0.21583060 -0.49649989  0.20596038 -0.17380545  0.11497834
## daphnia       -0.05623810 -0.84707187  0.12106895 -0.27778376  0.74460591
## sididae       -0.32799544 -0.18310338  0.57795933  0.01439421 -0.06343889
##                 copepod     daphnia     sididae
## ceriodaphnia  0.2158306 -0.05623810 -0.32799544
## cyclopoida   -0.4964999 -0.84707187 -0.18310338
## rotifer       0.2059604  0.12106895  0.57795933
## calanoid     -0.1738054 -0.27778376  0.01439421
## bosmina       0.1149783  0.74460591 -0.06343889
## copepod       1.0000000  0.74474618 -0.12982270
## daphnia       0.7447462  1.00000000 -0.06137393
## sididae      -0.1298227 -0.06137393  1.00000000
```

3) Data manipulation assignment: NYC flights

```r
library(nycflights13)
```

- Inspect the data

```r
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## # ... with 11 more variables: 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
str(flights)
```

```
## 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
# There are 3 airports in NYC
flights$origin%>%
  unique()
```

```
## [1] "EWR" "LGA" "JFK"
```

```r
#And a total of 105 destinations
flights$dest%>%
  unique()%>%
  length()
```

```
## [1] 105
```

Question 1 a) Which NYC airport has the most flights?

- Because each observation (ie. each row) represents a single flight, we simply tally the number of observation by each airport (using group_by() and summarise())

```
flights%>%
  group_by(origin)%>%
  summarise(count=n())
```

```
## # A tibble: 3 x 2
##   origin  count
##   <chr>   <int>
## 1 EWR    120835
## 2 JFK    111279
## 3 LGA    104662
```

- Answer for a) is: EWR > JFK > LGA

b) Which NYC airport flies to the most destinations?

- This is similar to a) but we now want to remove the duplicated destinations within each NYC airport (within each group) and we can do that easily by using the function distinct()

```
total_flights <- flights%>%
  group_by(origin)%>%
  distinct(dest)%>%
  summarise(count=n())
total_flights
```

```
## # A tibble: 3 x 2
##   origin count
##   <chr>  <int>
## 1 EWR       86
## 2 JFK       70
## 3 LGA       68
```

- Answer to B: EWR > JFK > LGA

Any alternative solutions??

c) BONUS: What are the top 3 destinations of each airport

- Option 1: split each airport up and get the two individually
- Option 2: Use pivot to keep the data together

```
#group_by summary to get the total flights to each dest by airport, then pivot
pivot_flight <- flights%>%
  group_by(origin,dest)%>%
  summarise(count=n())%>%
  pivot_wider(names_from='origin',values_from='count') #pivot
head(pivot_flight)
```

```
## # A tibble: 6 x 4
##   dest    EWR   JFK   LGA
##   <chr> <int> <int> <int>
## 1 ALB     439    NA    NA
## 2 ANC       8    NA    NA
## 3 ATL    5022  1930 10263
## 4 AUS     968  1471    NA
## 5 AVL     265    NA    10
```

```
## 6 BDL      443     NA     NA
```

- Then use the arrange() function to order each column in descending order

```r
#Arrange
pivot_flight%>%
  arrange(desc(EWR))
```

```
## # A tibble: 105 x 4
##    dest    EWR   JFK   LGA
##    <chr> <int> <int> <int>
##  1 ORD    6100  2326  8857
##  2 BOS    5327  5898  4283
##  3 SFO    5127  8204    NA
##  4 CLT    5026  2870  6168
##  5 ATL    5022  1930 10263
##  6 MCO    4941  5464  3677
##  7 LAX    4912 11262    NA
##  8 IAH    3973   274  2951
##  9 FLL    3793  4254  4008
## 10 DTW    3178  1166  5040
## # ... with 95 more rows
```

```r
pivot_flight%>%
  arrange(desc(JFK))
```

```
## # A tibble: 105 x 4
##    dest    EWR   JFK   LGA
##    <chr> <int> <int> <int>
##  1 LAX    4912 11262    NA
##  2 SFO    5127  8204    NA
##  3 BOS    5327  5898  4283
##  4 MCO    4941  5464  3677
##  5 SJU    1067  4752    NA
##  6 FLL    3793  4254  4008
##  7 LAS    2010  3987    NA
##  8 BUF     973  3582   126
##  9 MIA    2633  3314  5781
## 10 DCA    1719  3270  4716
## # ... with 95 more rows
```

```r
pivot_flight%>%
  arrange(desc(LGA))
```

```
## # A tibble: 105 x 4
##    dest    EWR   JFK   LGA
##    <chr> <int> <int> <int>
##  1 ATL    5022  1930 10263
##  2 ORD    6100  2326  8857
##  3 CLT    5026  2870  6168
##  4 MIA    2633  3314  5781
##  5 DTW    3178  1166  5040
##  6 DFW    3148   732  4858
##  7 DCA    1719  3270  4716
##  8 BOS    5327  5898  4283
##  9 FLL    3793  4254  4008
## 10 MSP    2377  1095  3713
```

```
## # ... with 95 more rows
```

Question 2 For simplicity, let's assume that delay means that there are delays in arrival and departure (ie. arrival>0 depart>0).

```
#filter observations with delays in both arrival and departure
delay_flights <- flights%>%
  filter(arr_delay>0&dep_delay>0)
```

a) Which airport has the "MOST" (ie. frequency) delays?

```
airport_delays <- delay_flights%>%
  group_by(origin)%>%
  summarise(delays=n())
```

b) Does the ranking of a) change after dividing by the number of flight for each airport (1a)?

- This is where ***join*** functions become useful

```
delay_ratio <- airport_delays %>%
  left_join(total_flights,by='origin')%>%
  mutate(ratio = delays/count)

data.frame(origin=delay_ratio$origin,DelayRatio=delay_ratio$ratio)
```

```
##   origin DelayRatio
## 1    EWR   438.0465
## 2    JFK   418.9000
## 3    LGA   372.1765
```

c) On average, which carrier has the "LONGEST" (ie. duration) delays? (Add arrival and departure delays together)

- Use mutate to combine variables

```
delay_flights%>%
  mutate(total_delay = arr_delay + dep_delay)%>%
  group_by(carrier)%>%
  summarise(mean_delay=mean(total_delay))%>%
  arrange(desc(mean_delay))
```

```
## # A tibble: 16 x 2
##    carrier mean_delay
##    <chr>        <dbl>
##  1 HA           148
##  2 OO           139.
##  3 9E           124.
##  4 YV           124.
##  5 F9           121.
##  6 EV           117.
##  7 VX           113.
##  8 AA           106.
##  9 DL           105.
## 10 MQ           105.
## 11 B6           101.
## 12 FL            99.2
## 13 WN            97.2
## 14 AS            96.1
## 15 UA            88.4
```

```
## 16 US             85.1
```

Question 3 Using the overall mean, convert travel distances into 2 distance categories (ie.longer or shorter than average). Do departure or arrival delay times differ betweeen distance categories?

- use ifelse() within the mutate function to assign distance categories based on the mean
- Then, calculate another set of averages (and SD if you want) of each category. Do this calculation for arrival delay and departure delay separtely

```
delay_flights%>%
  mutate(dist_class = ifelse(distance>mean(distance),'long','short'))%>%
  group_by(dist_class)%>%
  summarise(arrival_delay = mean(arr_delay),departure_delay = mean(dep_delay),
            arrival_sd = sd(arr_delay), departure_sd = sd(dep_delay))
```

```
## # A tibble: 2 x 5
##   dist_class arrival_delay departure_delay arrival_sd departure_sd
##   <chr>              <dbl>           <dbl>      <dbl>        <dbl>
## 1 long                49.2            47.8       58.6         58.7
## 2 short               54.6            53.9       59.9         59.4
```

4) Visualizing data with ggplot2

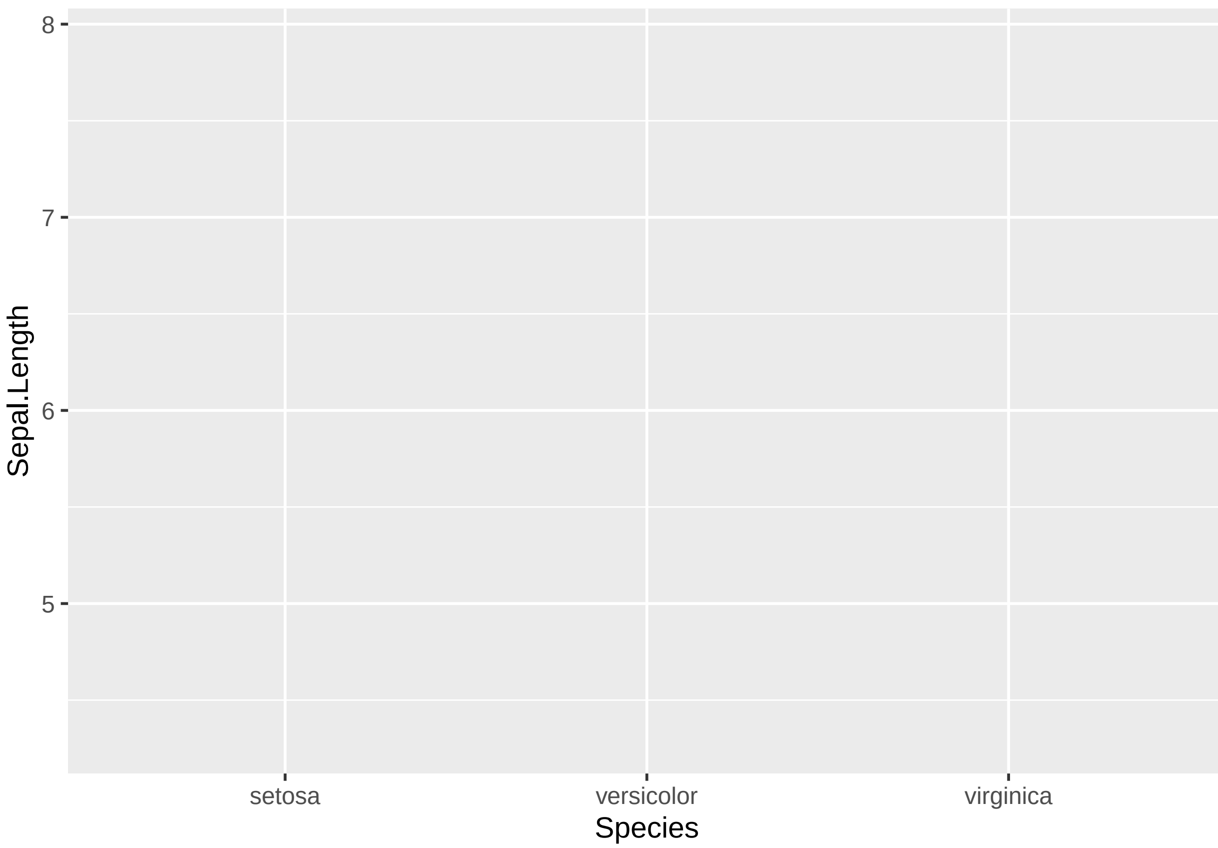
ggplot2 is the plotting package of tidyverse. It looks a little complicated at first but once you get used to it you'll realize that it is quite intuitive. Personally, I like it more than base R plotting functions because it is easy to customize, makes complex data easy to visualize, and more importantly, ggplot outputs can be stored into variables (unlike base R), making it extremely convenient for specific tasks such exploratory data analysis.

2a. Basic syntax

When plotting with ggplot, the first thing you have to do is to let it know which ***dataframe*** your data is coming form, which is your ***x*** and which is ***y***.

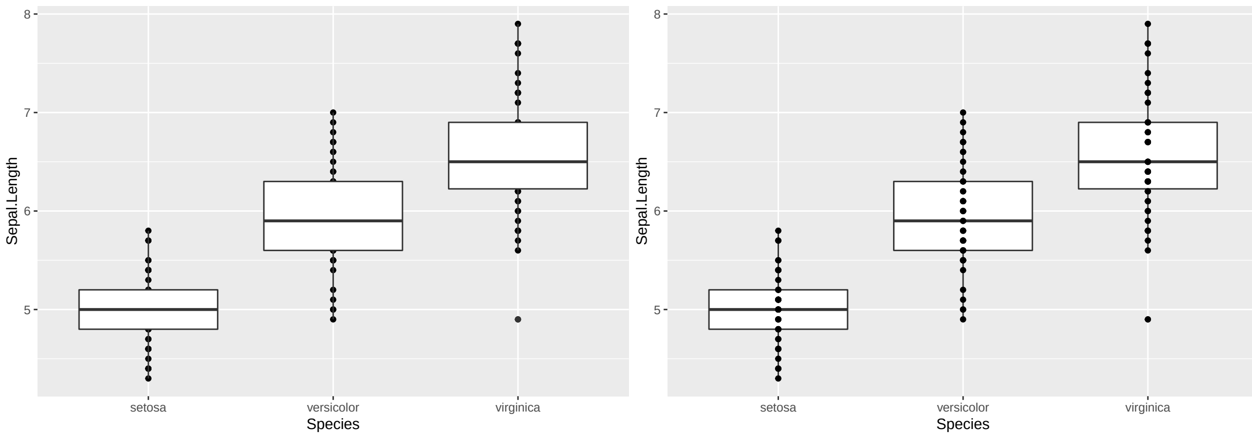
Using iris dataset as an example:

```
#To define the dataframe, its x's and y's, this is all you have to do.
ggplot(data=iris, aes(x=Species,y=Sepal.Length)) #aes() stands for aesthetics
```



As you can see, the previous line of code essentially sets up your graphing area with your specified x and y. But there is no points or anything plotted to it yet! To do so, you simply add (literally with "+" sign) additional functions to it.

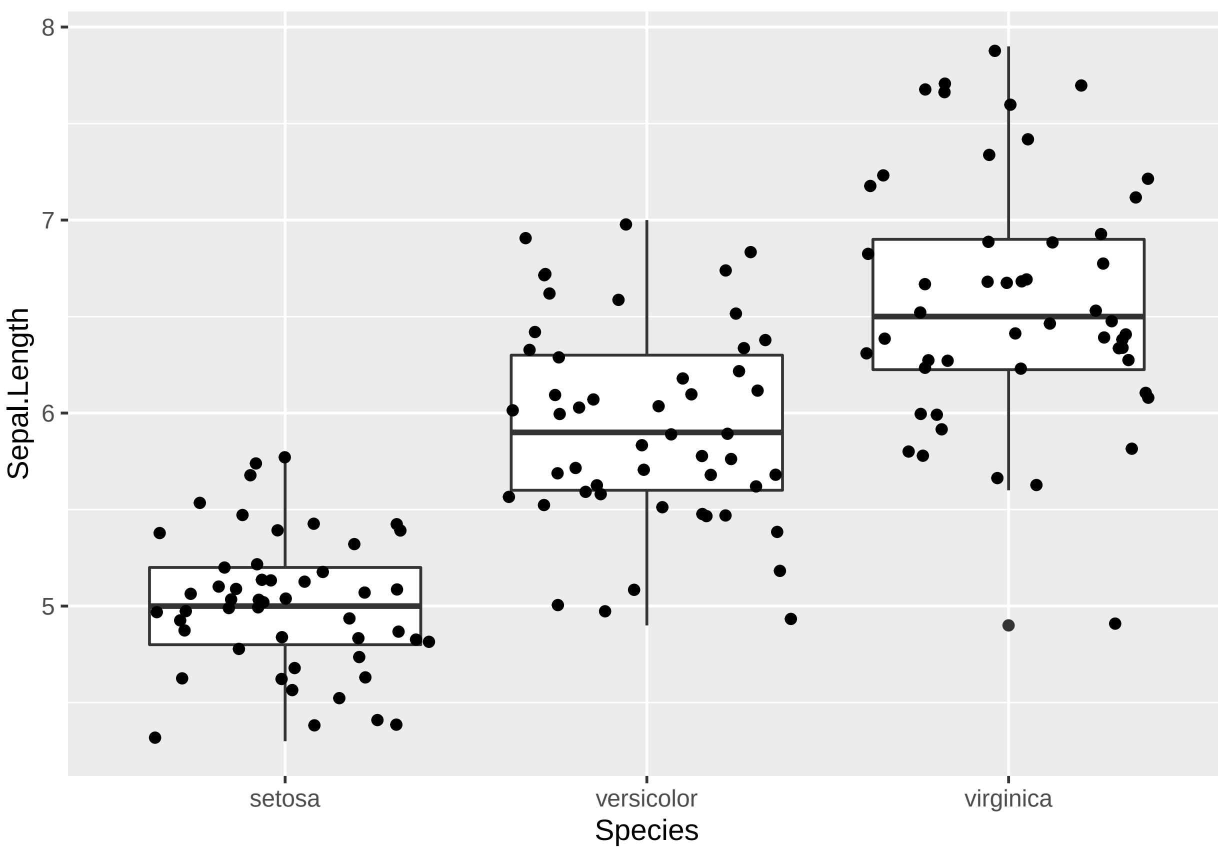
```
ggplot(iris,aes(x=Species,y=Sepal.Length))+geom_point()+geom_boxplot() #order matters

ggplot(iris,aes(x=Species,y=Sepal.Length))+geom_boxplot()+geom_point()#plots boxes first then points
```



You might have noticed that this grammar is very similar to the %>% operator we've seen previously. The idea is exactly the same. What you have set as your dataframe, and the x and y, gets passed down to the functions that you add to the initial ggplot(). That's why you do not need any additional arguments within in geom_point() and geom_boxplot().

As I have mentioned previously, ggplot objects can be stored in a variable. You can then "add" additional things to to it as you would normally do with any ggplot functions. Below is an example where I assign the ggplot object into a variable called "iris.p". I then add geom_jitter() to the iris.p and a plot containing iris.p with points jittered along the x-axis. Jittering is a way of spreading out points that are clustered together. This make easier to see how many points are actually there and they vary along the y-axis.

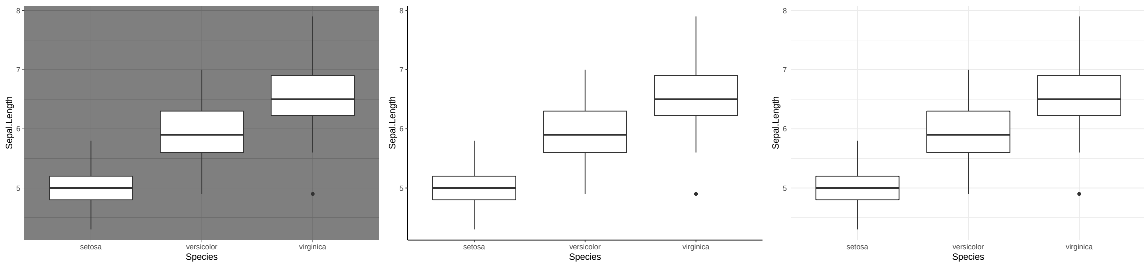
```
iris.p <- ggplot(iris,aes(x=Species,y=Sepal.Length))+geom_boxplot()
iris.p + geom_jitter()
```



2b. Customizing ggplot layout

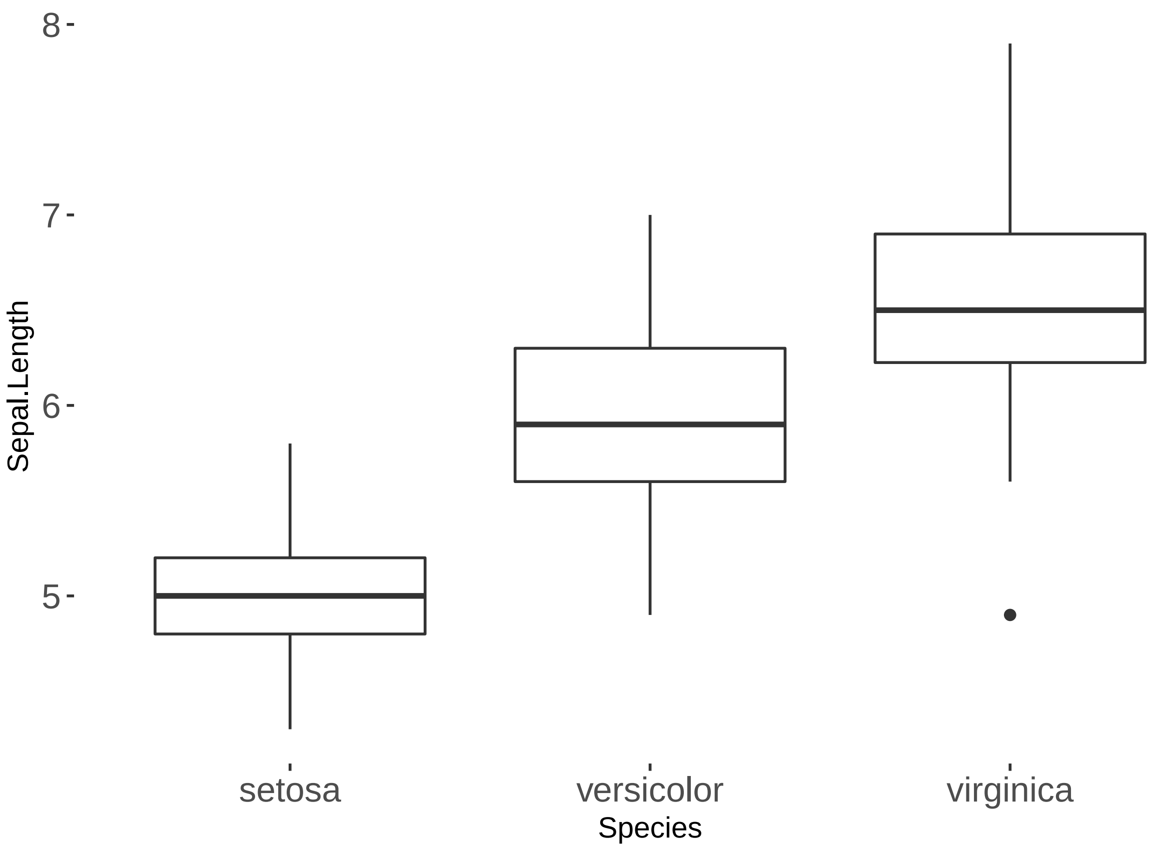
ggplot has a lot of built-in layouts, or "themes", ready to use on the spot. Here are some examples.

```
iris.p + theme_bw()
iris.p + theme_classic()
iris.p + theme_light()
```



These built-in themes will usually suffice but sometimes you might want to adjust the font sizes, especially for presentations. You can easily do so using the theme() function.

```
iris.p + theme(axis.text.x=element_text(size=13),
               axis.text.y=element_text(size=13),
               panel.background=element_blank())
```



You can probably tell which argument is for what feature of the plot. That's what I mean when I say ggplot (and tidyverse in general) is quite intuitive. You just have to familiarize yourself with them!

2c. Animations

You can even make animated figures pretty easily

```
#generate fake data
fake_data <- data.frame(y=rep(runif(45)),group=rep(c('a','b','c'),each=15),time=rep(seq(15),3))

#visualize
fake_p <-ggplot(fake_data,aes(x=time,y=y,group=group,col=group))+geom_point()+geom_line()

#animate
library(gganimate)
fake_p + transition_reveal(time)
```

Exercise: Make a figure with the nycflights dataset. You can plot the raw data or some some summary statistics of your choice.