Dplyr, Pipes, and More

Statistical Computing, STA3005

Jan 22, 2024

Last chapter: Purrr and a bit of dplyr

- For iteration tasks, `plyr` and `tidyverse` are two alternative of base R with better consistency
- `plyr` provides apply-like functions `**ply` and ensures the consistency of input and output data types
- `tidyverse` is a collection of packages for common data science tasks
- `purrr` is one such package that provides a consistent family of iteration functions
- Compared with `plyr`, `purrr` is often faster
- `map()` : list in, list out
- `map_dbl()`, `map_lgl()`, `map_chr()` : list in, vector out (of a particular data type)
- `map_dfr()`, `map_dfc()` : list in, data frame out (row-binded or column-binded)
- `dplyr` is another such package that provides functions for data frame computations
- `filter()` : subset rows based on a condition
- `group_by()` : define groups of rows according to a condition
- `summarize()` : apply computations across groups of rows

Part I: *Revisit tidyverse*

What is the tidyverse?

The tidyverse is a coherent collection of packages in R for data science, and `tidyverse` is itself a actually package that loads all its constituent packages. Packages include:

- **Data wrangling**: `dplyr`, `tidyr`, `readr`
- **Iteration**: `purrr`
- **Visualization**: `ggplot2`

Last chapter we covered `purrr` and a bit of `dplyr`. This chapter we'll do more `dplyr`, some `tidyr` and basic `ggplot2`.

Loading the tidyverse so that we can get all this functionality:

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.2.3
```

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```
## Warning: package 'tibble' was built under R version 4.2.3
```

```
## Warning: package 'tidyr' was built under R version 4.2.3
```

```
## Warning: package 'readr' was built under R version 4.2.3
```

```
## Warning: package 'dplyr' was built under R version 4.2.3
```

```
## Warning: package 'forcats' was built under R version 4.2.3
```

```
## Warning: package 'lubridate' was built under R version 4.2.3
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.3     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.1
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the ]8;;http://conflicted.r-lib.org/ conflicted package ]8;; to force all conflict
```

Why the tidyverse?

- Packages have a very consistent API
- Very active developer and user community
- Function names and commands follow a focused **grammar**
- Powerful and fast when working with data frames and lists (matrices, not so much, yet!)
- Pipes (`%>%` operator) allows us to fluidly glue functionality together
- At its best, tidyverse code can be **read like a story** using the pipe operator!

CRAN R Packages by Number of Downloads

UPDATED DAILY. Last updated: 2023-01-19 01:19:17 +1100 Melbourne time (about 10 hours ago)

Rank	Package Name	Downloads	Author	Maintainer
1	magrittr	105,620,859		
2	ggplot2	103,494,953		
3	rlang	96,844,031		
4	dplyr	80,668,370		
5	tibble	67,814,202		
6	jsonlite	67,626,269		
7	Rcpp	67,007,649		
8	vctrs	66,101,582		
9	pillar	63,363,428		
10	devtools	63,052,277		
11	glue	62,663,280		
12	cli	62,517,641		
13	stringr	55,984,602		
14	stringi	55,503,730		
15	lifecycle	55,377,940		
16	aws.ec2metadata	54,943,359		
17	digest	54,043,682		
18	rsconnect	51,654,445		
19	knitr	50,700,992		
20	aws.s3	49,296,067		
21	tidyverse	48,758,134		
22	tidyr	48,748,390		

Screenshot from http://www.datasciencemeta.com/rpackages.

The developer of `tidyverse` package, Hadley Wickham, wined the 2019 COPSS Presidents' Award (one of the highest honors in statistics). The award citation recognized Wickham "for influential work in statistical computing, visualization, graphics, and data analysis; for developing and implementing an impressively comprehensive computational infrastructure for data analysis through R software; for making statistical thinking and computing accessible to large audience; and for enhancing an appreciation for the important role of statistics among data scientists."

Data wrangling the tidy way

- Packages `dplyr` and `tidyr` are going to be our main workhorses for data wrangling
- Main structure these packages use is the data frame (or tibble, but we won't go there)
- Learning pipes `%>%` will facilitate learning the `dplyr` and `tidyr` functions
- `dplyr` functions are analogous to Structured Query Language (SQL) counterparts, so learn `dplyr` and get SQL for free!

Part II: *Mastering the pipe*

All behold the glorious pipe

- Tidyverse functions are at their best when composed together using the pipe operator
- It looks like this: `%>%`. **Shortcut**: use `ctrl + shift + m` in RStudio
- This operator actually comes from the `magrittr` package (the most downloaded package, automatically included in `dplyr`)
- **Piping** at its most basic level:

> Take one return value and automatically feed it in as an input to another function, to form a flow of results

- In unix and related systems, we also have pipes, as in:

```
ls -l | grep tidy | wc -l
```

List the detailed information of files and folders in the current directory, **and then** keep lines containing `tidy`, **and then** count the number of lines. Finally, it outputs the number of files or directories whose names contain `tidy`.

How to read pipes: single arguments

Passing a single argument through pipes, we interpret something like:

```
x %>% f %>% g %>% h
```

as `h(g(f(x)))`

Key takeaway: in your mind, when you see `%>%`, read this as **"and then"**

Simple example

We can write `exp(1)` with pipes as `1 %>% exp`, and `log(exp(1))` as `1 %>% exp %>% log`

```
exp(1)
```

```
## [1] 2.718282
```

```
1 %>% exp
```

```
## [1] 2.718282
```

```
1 %>% exp %>% log
```

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

How to read pipes: multiple arguments

Now for multi-arguments functions, we interpret something like:

```
x %>% f(y)
```

as `f(x,y)`

Simple example

```
mtcars %>% head(4)
```

And what's the "old school" (base R) way?

```
head(mtcars, 4)
```

Notice that, with pipes:

- Your code is more readable (arguably)
- You can run partial commands more easily

The dot (起占位符作用)

The command `x %>% f(y)` can be equivalently written in **dot notation** as:

```
x %>% f(., y)
```

Advantage of using dots: Sometimes you want to pass in a variable as the *second* or *third* (say, not first) argument to a function, with a pipe. As in:

```
x %>% f(y, .)
```

which is equivalent to `f(y,x)`

Simple example

Again, see if you can interpret the code below without running it, then run it in your R console as a way to check your understanding:

```
state_df = data.frame(state.x77)
state.region %>%
  tolower %>%
  tapply(state_df$Income, ., summary)
```

(income level of each region)

A more complicated example:

```
x = "We learn piping from this chapter"
x %>%
  strsplit(split = " ") %>%
  .[[1]] %>% # indexing, could also use `[[`(1)
  nchar %>%
  max
```

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

Part III: *dplyr functions*

Some of the most important `dplyr` functions:

- `filter()`: subset rows based on a condition
- `group_by()`: define groups of rows according to a condition
- `summarize()`: apply computations across groups of rows
- `arrange()`: order rows by value of a column
- `select()`: pick out given columns
- `mutate()`: create new columns
- `mutate_at()`: apply a function to given columns

We've learned `filter()`, `group_by()`, `summarize()` in the last chapter.

In the following, we take the data frame `mtcars` as an example. It stores the fuel consumption and 10 design and performance features of 32 US automobiles in 1974.

```
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
```

For example, the first column `mpg` stands for miles per gallon.

`arrange()`: order rows by values of a column

By default, `arrange()` runs in ascending order: (由小到大排)

```
mtcars %>%
  arrange(mpg) %>%
  head(4)
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
```

类似于 base R 中的 order，但 order 返回的是 index vector

```
# Base R
mpg_inds = order(mtcars$mpg)
head(mtcars[mpg_inds, ], 4)
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
```

Compared with base R, piping of `dyplr` is tidier and more compact.

We can apply in descending order:

```
mtcars %>%
  arrange(desc(mpg)) %>%
  head(4)
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

```
# Base R
mpg_inds_decr = order(mtcars$mpg, decreasing = TRUE)
head(mtcars[mpg_inds_decr, ], 4)
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

We can order by multiple columns at the same time: 可以用于对各个类别内部分别排序

```
mtcars %>%
  arrange(desc(gear), desc(hp)) %>%
  head(8)
```

gear 相同时再按 hp 排

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
```

First, the data frame is ordered by `gear` column; if there is a tie in `gear`, then ordered by `hp` column.

`select()`: pick out given columns

Here, we pick out three columns— `cyl` for number of cylinders, `disp` for displacement and `hp` for horsepower—from the original data frame.

```
mtcars %>%
  select(cyl, disp, hp) %>%
  head(2)
```

```
##               cyl disp  hp
## Mazda RX4       6  160 110
## Mazda RX4 Wag   6  160 110
```

```
# Base R
head(mtcars[, c("cyl", "disp", "hp")], 2)
```

```
##                cyl disp  hp
## Mazda RX4       6  160 110
## Mazda RX4 Wag   6  160 110
```

Some handy `select()` helpers

Select columns whose names start with "d".

```
mtcars %>%
  select(starts_with("d")) %>%
  head(2)
```

```
##                disp drat
## Mazda RX4       160  3.9
## Mazda RX4 Wag   160  3.9
```

```
# Base R
d_colnames = grep(x = colnames(mtcars), pattern = "^d")
head(mtcars[, d_colnames], 2)
```

```
##                disp drat
## Mazda RX4       160  3.9
## Mazda RX4 Wag   160  3.9
```

Base R replies on text manipulation function `grep` to search the entries starting with *d* in a string vector. (We will introduce more in the next chapter)

We can do many other things as well:

```
mtcars %>% select(ends_with('t')) %>% head(2)
```

```
##                drat    wt
## Mazda RX4       3.9 2.620
## Mazda RX4 Wag   3.9 2.875
```

```
mtcars %>% select(ends_with('yl')) %>% head(2)
```

```
##                cyl
## Mazda RX4       6
## Mazda RX4 Wag   6
```

```
mtcars %>% select(contains('ar')) %>% head(2)
```

```
##                gear carb
## Mazda RX4         4    4
## Mazda RX4 Wag     4    4
```

If you're interested go and read more here

`mutate()` : create one or several columns

```r
mtcars = mtcars %>%
  mutate(hp_wt = hp/wt,
         mpg_wt = mpg/wt)

# Base R
mtcars$hp_wt = mtcars$hp/mtcars$wt
mtcars$mpg_wt = mtcars$mpg/mtcars$wt
```

Newly created variables are usable immediately:

```r
mtcars = mtcars %>%
  mutate(hp_wt_again = hp/wt,
         hp_wt_cyl = hp_wt_again/cyl)

# Base R
mtcars$hp_wt_again = mtcars$hp/mtcars$wt
mtcars$hp_wt_cyl = mtcars$hp_wt_again/mtcars$cyl
```

`mutate_at()` : apply a function to one or several columns

```r
mtcars = mtcars %>%
  mutate_at(c("hp_wt", "mpg_wt"), log)      (overwrite columns)

# Base R
mtcars$hp_wt = log(mtcars$hp_wt)
mtcars$mpg_wt = log(mtcars$mpg_wt)
```

Important note

Calling `dplyr` functions always outputs a new data frame, it *does not alter* the existing data frame

Thus, to keep the changes, we have to reassign the data frame to be the output of the pipe! (Look back at the examples for `mutate()` and `mutate_at()`)

`dplyr` and SQL

SQL is commonly used in accessing, manipulating and retrieving databases.

- Once you learn `dplyr` you should find SQL very natural, and vice versa!
- For example, `select` is `SELECT`, `filter` is `WHERE`, `arrange` is `ORDER BY` etc.
- This will make it much easier for tasks that require using both R and SQL to munge data and build statistical models
- One major link is through powerful functions like `group_by()` and `summarize()`, which are used to aggregate data
- Another major link to SQL is through merging/joining data frames, via `left_join()` and `inner_join()` functions

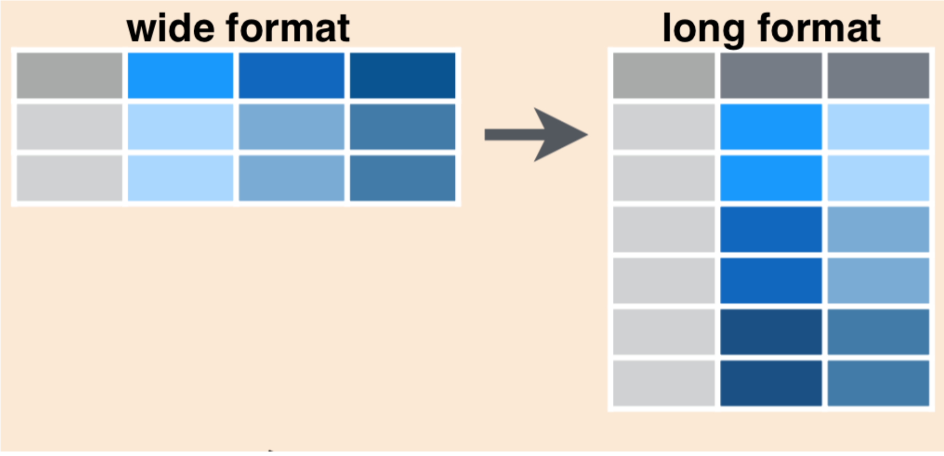
Part IV: *`tidyr` functions*

Two of the most important `tidyr` functions:

- `pivot_longer()` : make "wide" data longer
- `pivot_wider()` : make "long" data wider

There are many others like `spread()`, `gather()`, `nest()`, `unnest()`, etc. (If you're interested go and read about them here)

`pivot_longer()` : make "wide" data longer



`cases` is a 3 * 4 data frame, which stores the tuberculosis cases of three countries in 2011, 2012 and 2013.

```
library(EDAWR) # Load some nice data sets
cases
```

```
##   country  2011  2012  2013
## 1      FR  7000  6900  7000
## 2      DE  5800  6000  6200
## 3      US 15000 14000 13000
```

```
dim(cases)
```

```
## [1] 3 4
```

```
cases %>%
  pivot_longer(names_to = "year", # put column names into a column called "year"
               values_to = "n",   # put values into a column called "n"
               cols = 2:4) # where the values come from
```

```
## # A tibble: 9 × 3
##   country year      n
##   <chr>   <chr> <dbl>
## 1 FR      2011   7000
## 2 FR      2012   6900
## 3 FR      2013   7000
## 4 DE      2011   5800
## 5 DE      2012   6000
## 6 DE      2013   6200
## 7 US      2011  15000
## 8 US      2012  14000
## 9 US      2013  13000
```

等价于：

```
df_out <- NULL
for(i in 1:3){
  for(j in 1:3){
    row <- c("country" = cases[i,1],
             "year" = colnames(cases)[j+1],
             "cases" = cases[i, j+1])
    df_out <- rbind(df_out, row)
  }
}
```

等价于：

```
data.frame(country = rep(cases[,1], each = 3),
           year = rep(colnames(cases)[2:4], 3)
           n = as.vector(t(as.matrix(cases[,2:4]))))
```

可省略，t(data.frame) 会自动 coercion 成 matrix

- Here, we transposed columns 2:4 into a `year` column
- We put the corresponding count values into a column called `n`
- Note `tidyr` did all the heavy lifting of the transposing work
- We just had to declaratively specify the output

As `EDAWR` package is not published on CRAN, we need to install this package from GitHub.

```
install.packages("devtools")
devtools::install_github("rstudio/EDAWR")
```

```
# Different approach to do the same thing
cases %>%
  pivot_longer(names_to = "year",
               values_to = "n",
               -country)
```

不需要 quotation (除去 country 的 columns)

```
## # A tibble: 9 × 3
##   country year      n
##   <chr>   <chr> <dbl>
## 1 FR      2011   7000
## 2 FR      2012   6900
## 3 FR      2013   7000
## 4 DE      2011   5800
## 5 DE      2012   6000
## 6 DE      2013   6200
## 7 US      2011  15000
## 8 US      2012  14000
## 9 US      2013  13000
```

```
# Could also do:
# cases %>%
#   pivot_longer(names_to = "year",
#                values_to = "n",
#                c(`2011`, `2012`, `2013`))
```

`pivot_wider()` : make "long" data wider

`cases` is a 6 * 3 data frame. It stores the mean annual particle concentration of three cities in 2014. Particle concentration are measured in two ways, large (PM10) and small (PM2.5).

```
pollution
```

```
##       city  size amount
## 1 New York large     23
## 2 New York small     14
## 3   London large     22
## 4   London small     16
## 5  Beijing large    121
## 6  Beijing small     56
```

```
dim(pollution)
```

```
## [1] 6 3
```

```
pollution %>%
  pivot_wider(names_from = "size",
              values_from = "amount")
```

```
## # A tibble: 3 × 3
##   city     large small
##   <chr>    <dbl> <dbl>
## 1 New York    23    14
## 2 London      22    16
## 3 Beijing    121    56
```

← 若删除 pollution 的第6行 data，则这里会变成 NA

- Here we transposed to a wide format by size

可以使用 filter() 处理掉 missing value

- We tabulated the corresponding amount for each size
- Note `tidyr` did all the heavy lifting again
- We just had to declaratively specify the output
- Note that `pivot_wider()` and `pivot_longer()` are inverses

Part V: *`ggplot2` verbs*

`gg` in the package name stands for the grammar of graphics, which means the whole system and structure of a language for plotting In `ggplot2`. A graphic

- maps **data** to
- the **aesthetic attributes** (such as color, shape and size) of
- **geometric objects** (points, lines and bars),
- contains possible **statistical transformations** of the concerned data
- on the given **coordinate systems**
- and may also use **faceting**, which controls the subsets of the data included in the plot.

In other words, a particular graphic is comprised of the following six components:

- data
- layer
- scale
- coordinate
- facet
- theme

In particular, we combine the five components (except data) by `+`.

`iris` database gives the measurements in cm of sepal length/width and petal length/width for 50 flowers from 3 different species of iris.

```
head(iris)
```

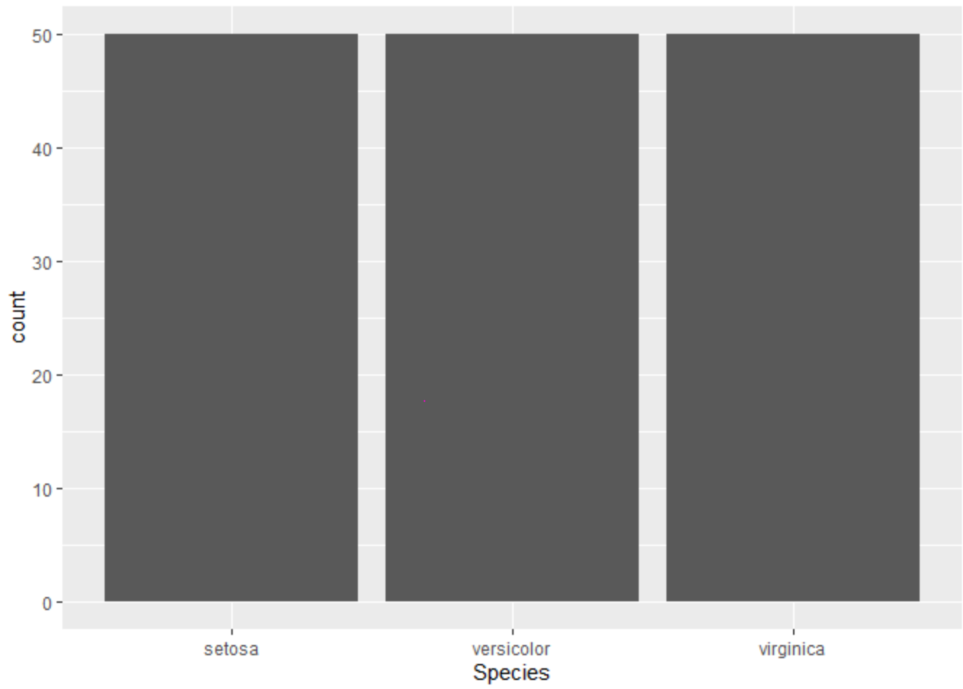
```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Mapping components: layer

- `geom`: geometric elements, such as bars, lines, points, etc.
- `stat`: statistical transformations, such as count and identity

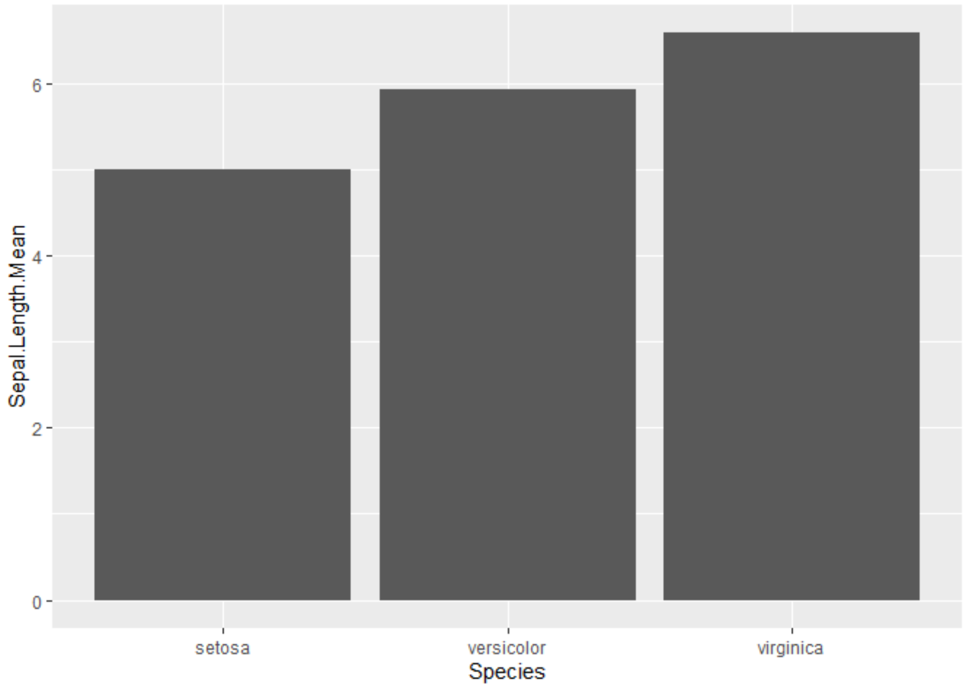
```
# Layer of coordinate system
iris %>%
  ggplot(aes(x = Species))
```


```r
# Layer of a bar plot
iris %>%
  ggplot(aes(x = Species)) +
  geom_bar(stat = "count")
```



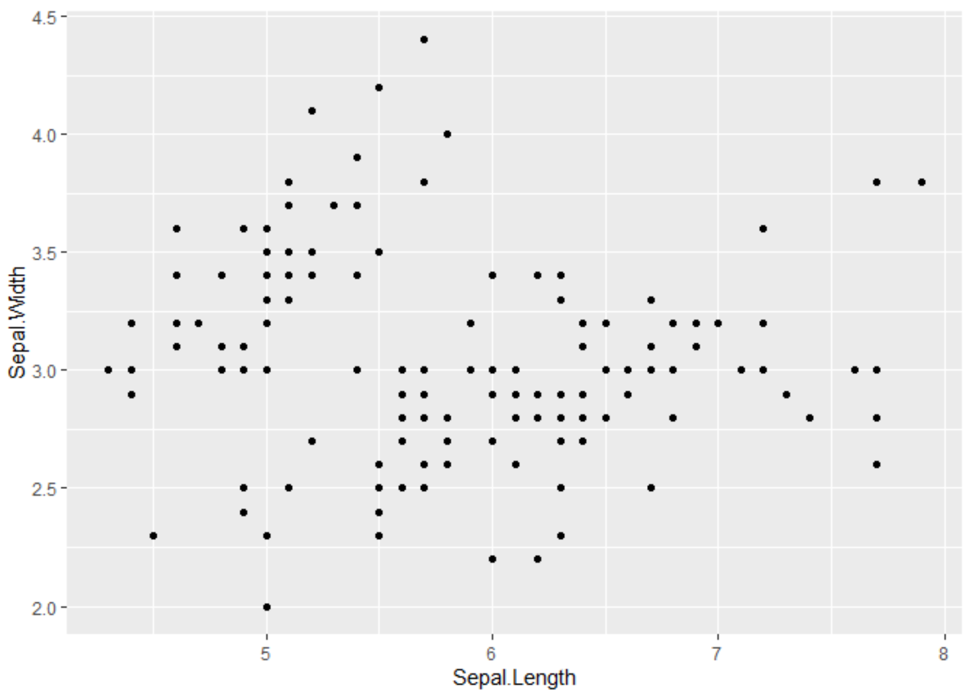

```r
# Barplot of mean sepal length
iris %>%
  group_by(Species) %>%
  summarize(Sepal.Length.Mean = mean(Sepal.Length)) %>%
  ggplot(aes(x = Species, y = Sepal.Length.Mean)) +
  geom_bar(stat = "identity") # what happens if you don't include the stat argument?
```

会报错，默认为 stat = "count" ⇒ 只能有一个 x 或 y aesthetic



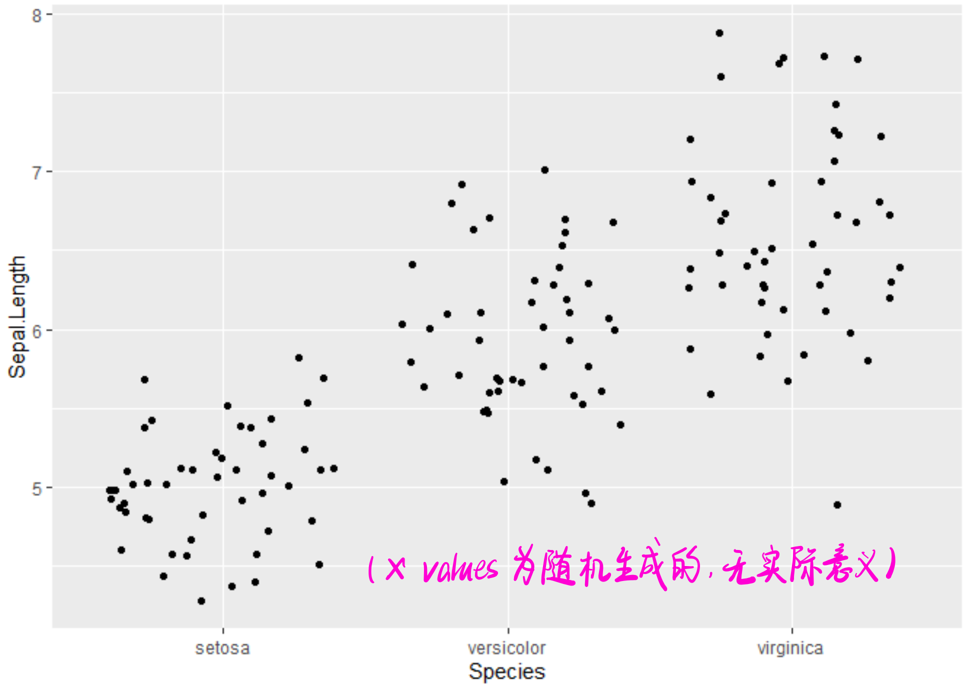

```r
# Scatter plot of sepal length vs width
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()
```

ggplot 需要 long format 来画 scatter plot

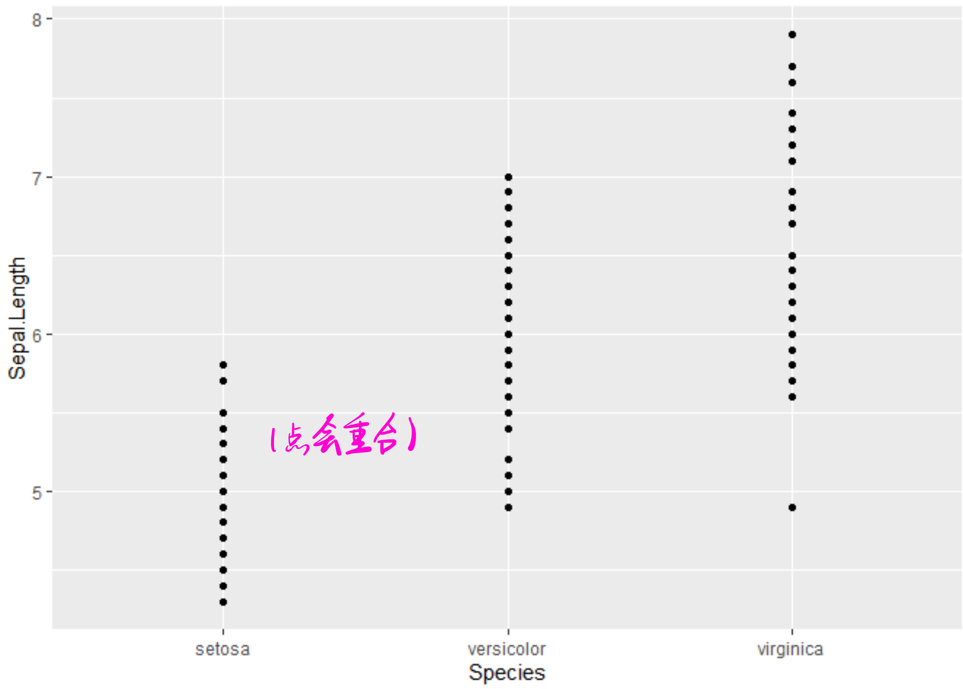



```r
# Jitter plot
iris %>%
  ggplot(aes(x = Species, y = Sepal.Length)) +
  geom_point(position = "jitter")
```

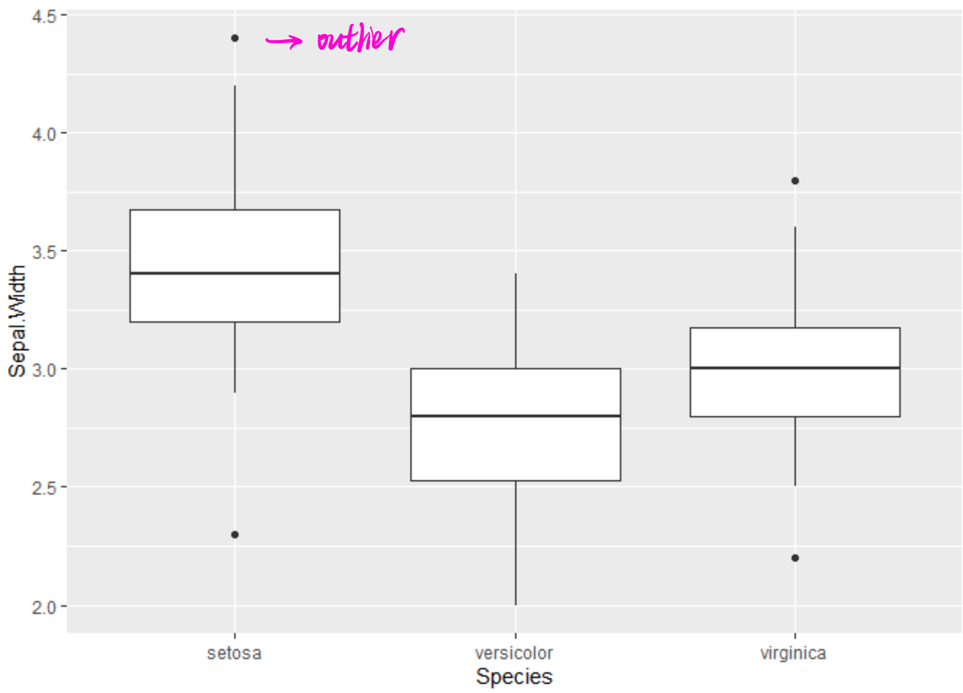
(适用于存在 discrete value 的情况)




```r
# without jittering
iris %>%
  ggplot(aes(x = Species, y = Sepal.Length)) +
  geom_point()
```




```r
# Boxplot
iris %>%
  ggplot(aes(x = Species, y = Sepal.Width)) +
  geom_boxplot()
```

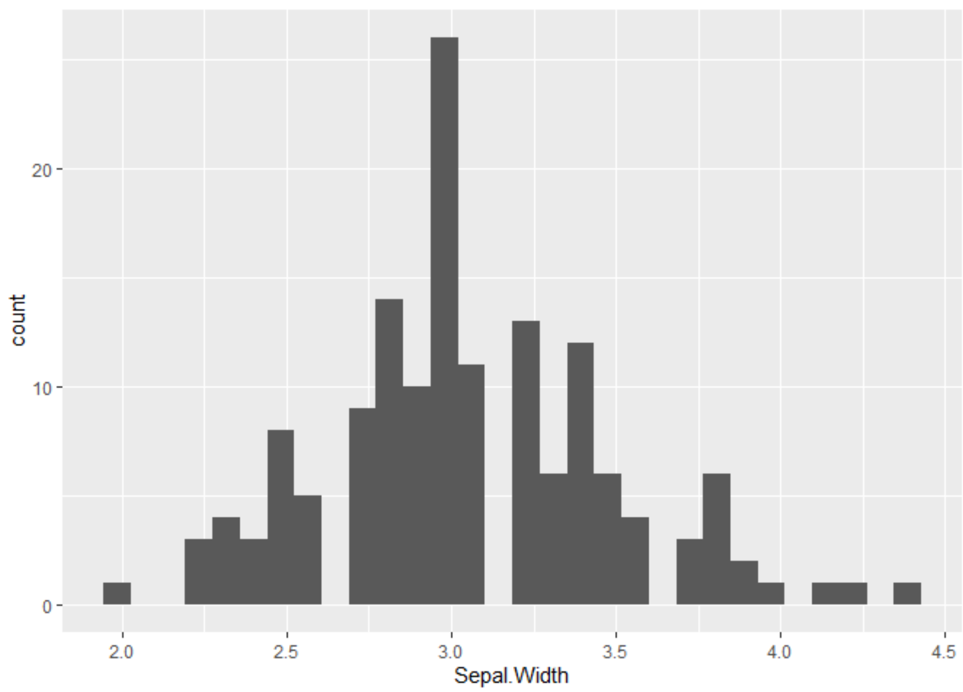


可以 combine 同一个 coordinate 下
不同的 layers
(如 jitter plot 和 boxplot)

```
# Histogram
iris %>%
  ggplot(aes(x = Sepal.Width)) +
  geom_histogram()
```

可以设置 `bins` 和 `binwidth`

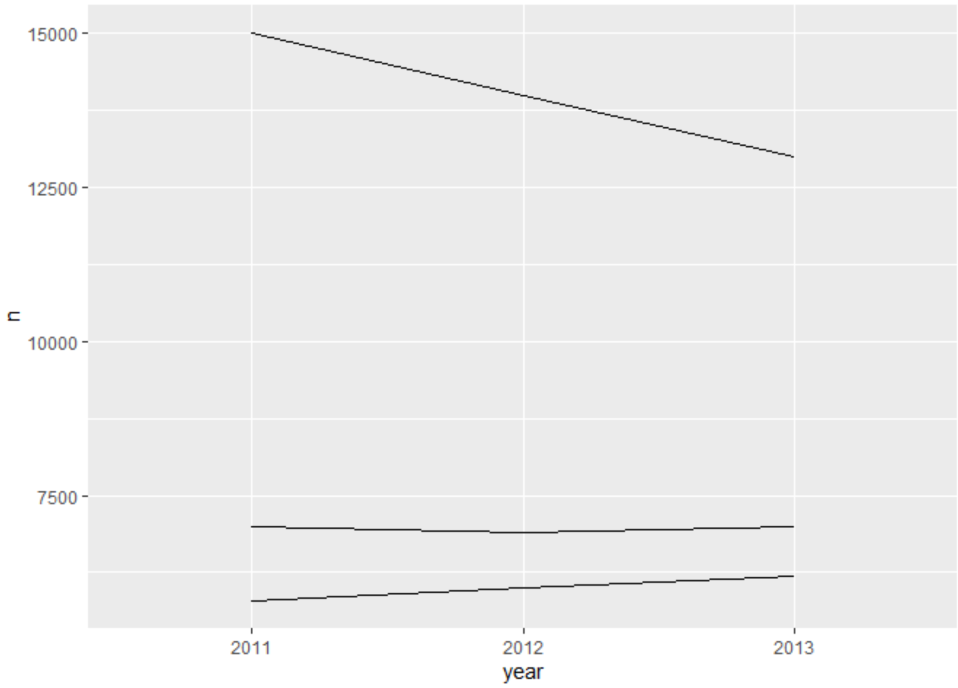
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
# Line chart
cases %>%
  pivot_longer(names_to = "year",
               values_to = "n",
               cols = 2:4) %>%
  ggplot(aes(x = year, y = n)) +
  geom_line(aes(group = country))
```

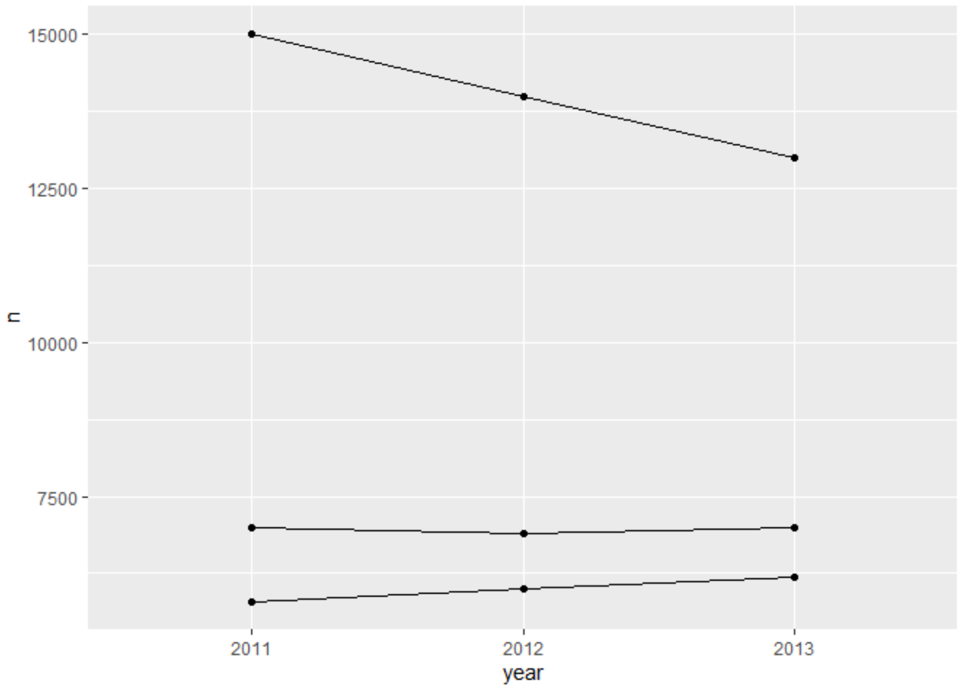
调整为 long format

不设置的话会 group by year，作出三条垂线




```
cases %>%
  pivot_longer(names_to = "year",
               values_to = "n",
               cols = 2:4) %>%
  ggplot(aes(x = year, y = n, group = country)) +
  geom_line() +
  geom_point()
```

位置改变了，类似于变成了 global variable



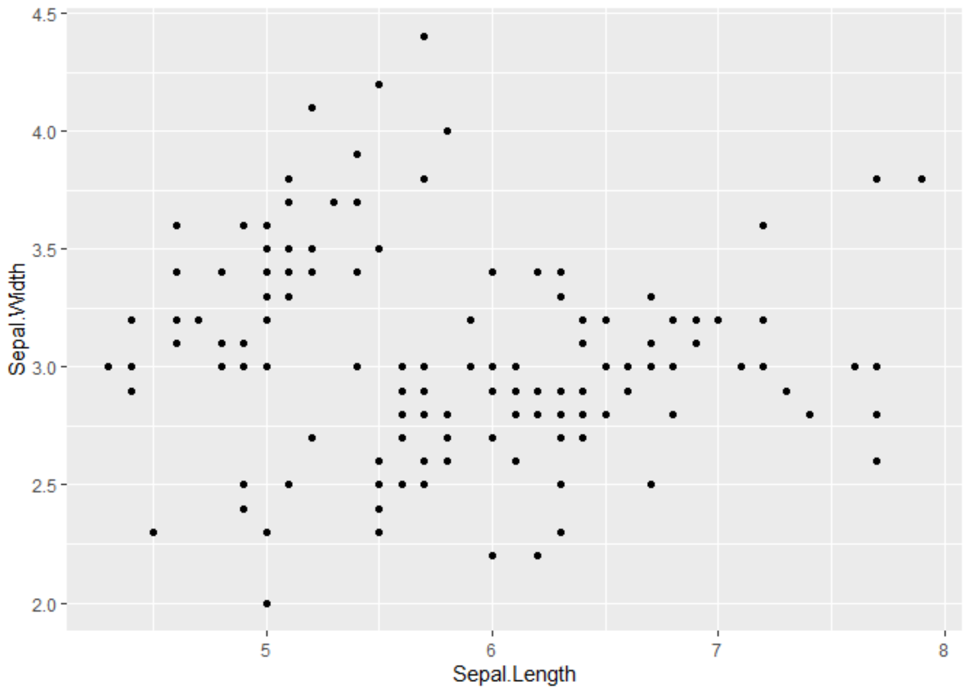
- Jittering of scatter plots reduces overlapping in the plots of ordinal data or data that are rounded

Mapping components: scale

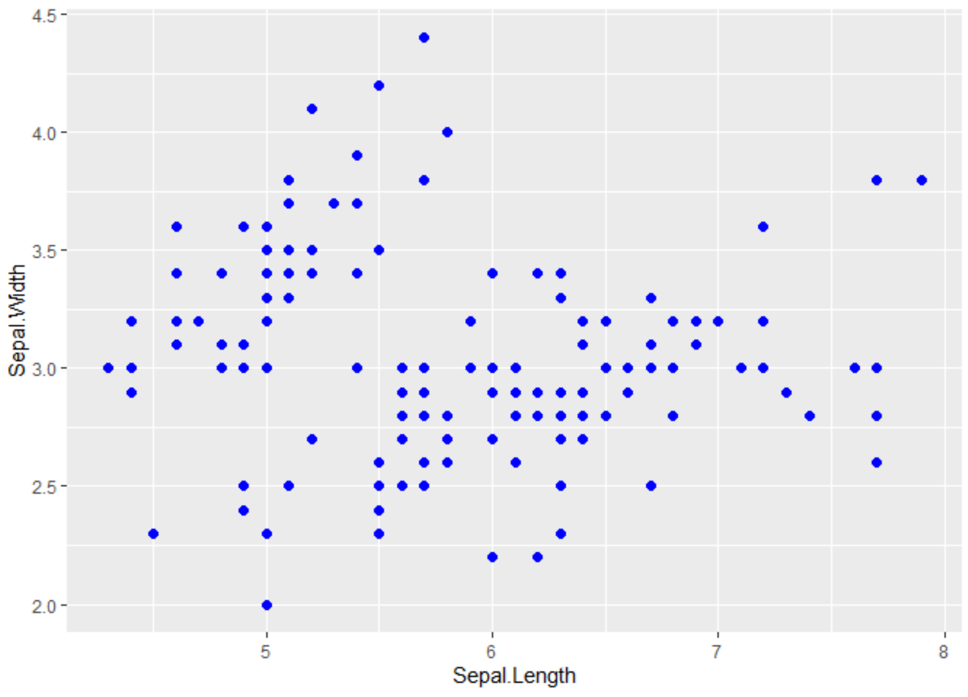
- Modify the aesthetic attributes of data, including shape, color, size, etc
- Adjust the scale of axis
- Add legends

```
# No aesthetic
iris %>%
```

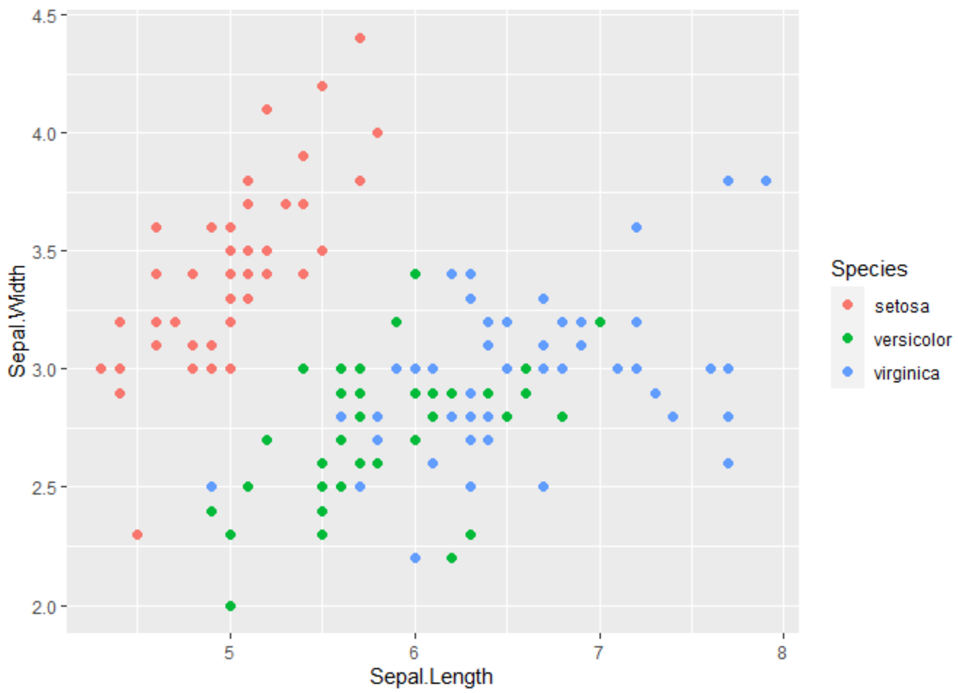
```
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point()
```



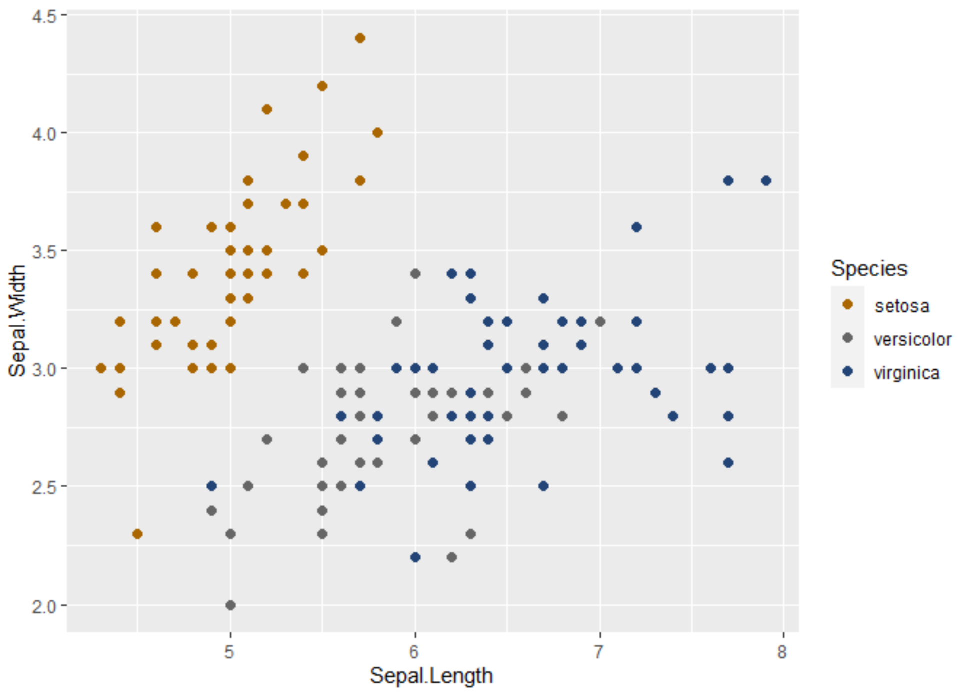

```
# Change point color and size
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(color = "blue", size = 2)
```




```
# Color points by group
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species), size = 2)
```

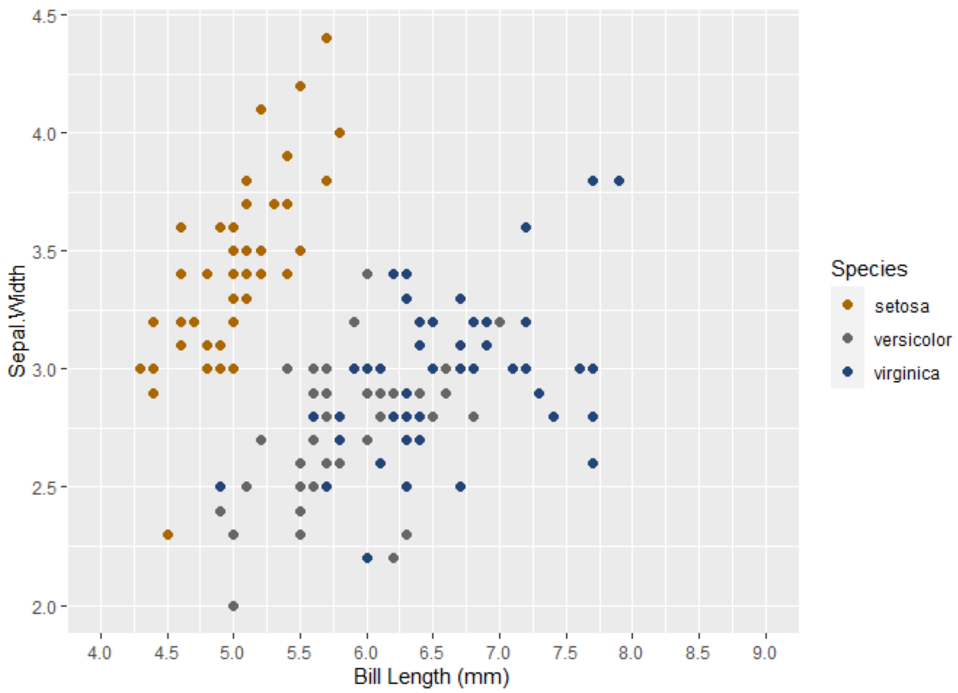



```
# Manually set the color for each group
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species), size = 2) +
  scale_color_manual(values = c("#aa6600","#666666","#224477"))
```

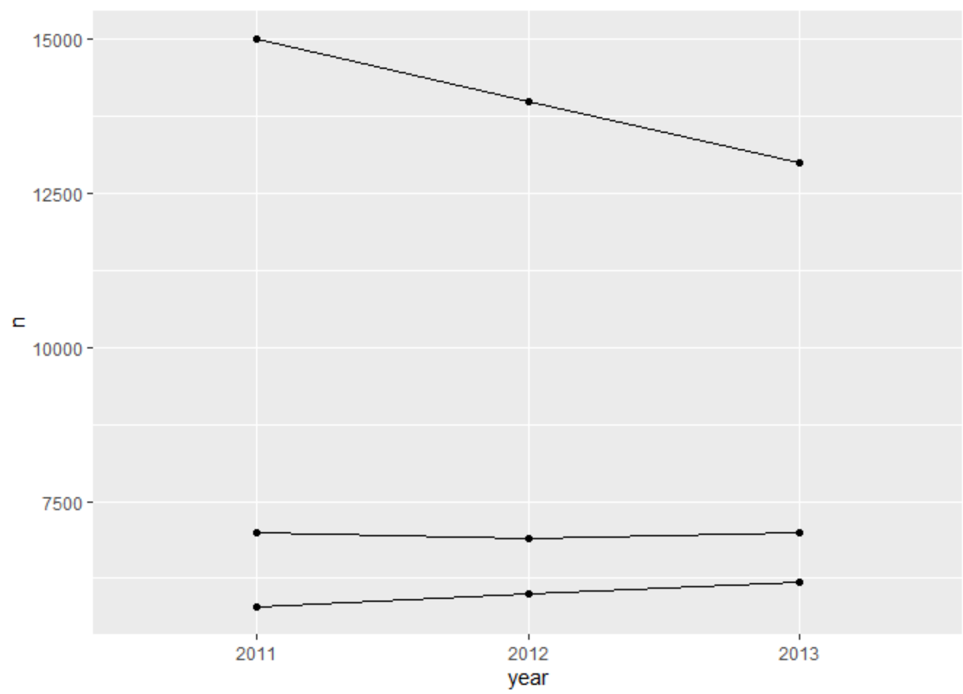



```
# Change the scale of axis
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species), size = 2) +
  scale_color_manual(values = c("#aa6600","#666666","#224477")) +
  scale_x_continuous(name = "Bill Length (mm)",
                     breaks = seq(4, 9, by = 0.5), limits = c(4, 9))
```

上/下界




```
# Previous line chart
cases %>%
  pivot_longer(names_to = "year",
               values_to = "n",
               cols = 2:4) %>%
  ggplot(aes(x = year, y = n, group = country)) +
  geom_line() +
  geom_point()
```

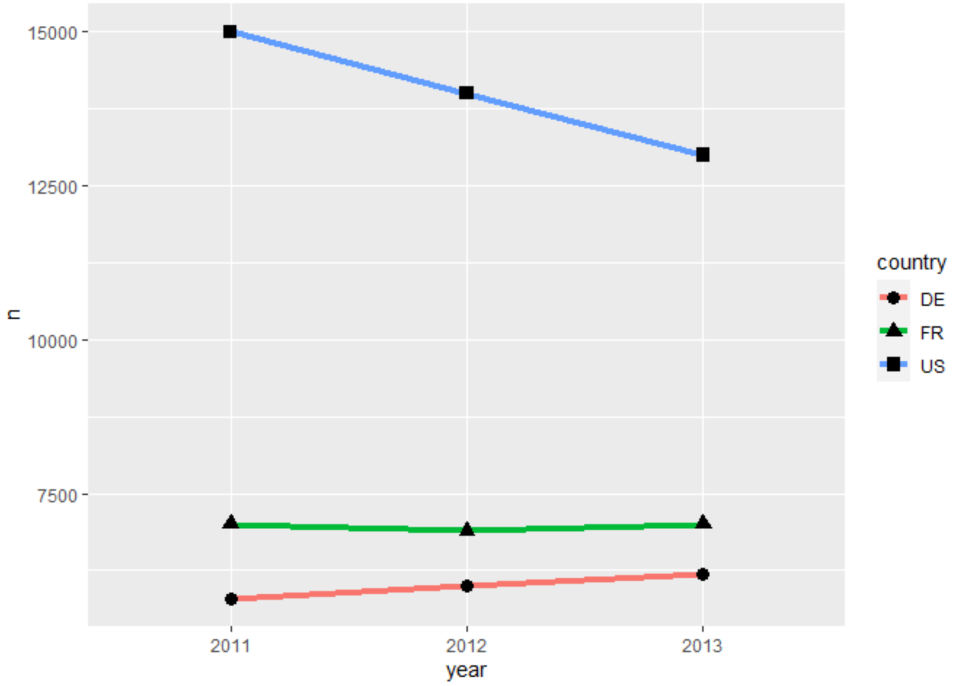



```
# Change the color and width for the line layer and the shape and size for the point layer
cases %>%
  pivot_longer(names_to = "year",
               values_to = "n",
               cols = 2:4) %>%
  ggplot(aes(x = year, y = n, group = country)) +
  geom_line(aes(color = country), size = 1.5) +
  geom_point(aes(shape = country), size = 3)
```

建议设置 linewidth

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
```

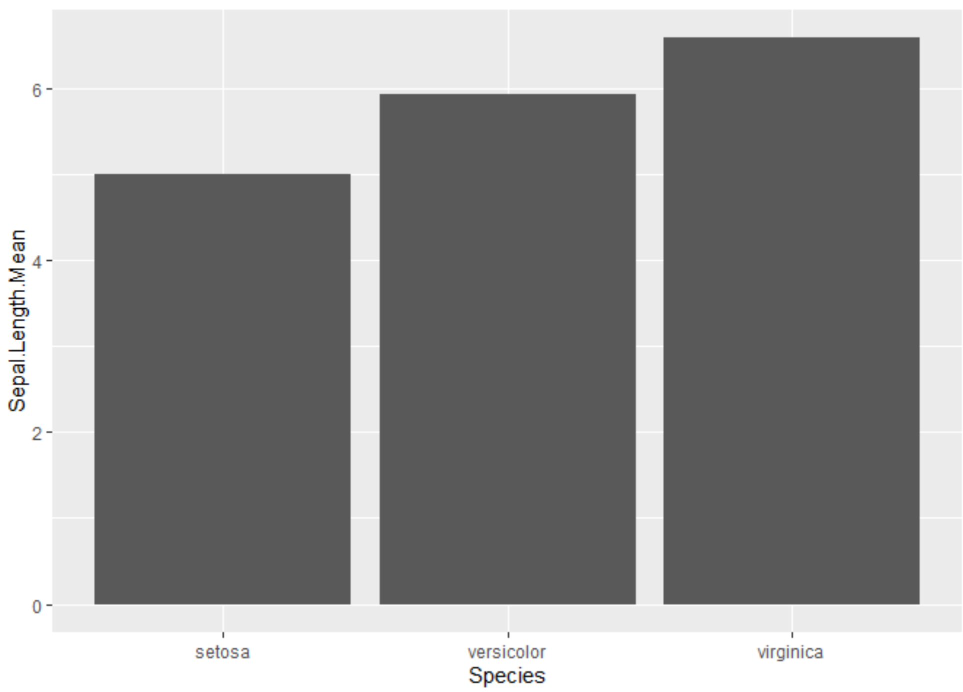
```
## ℹ Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



Mapping components: coordinate

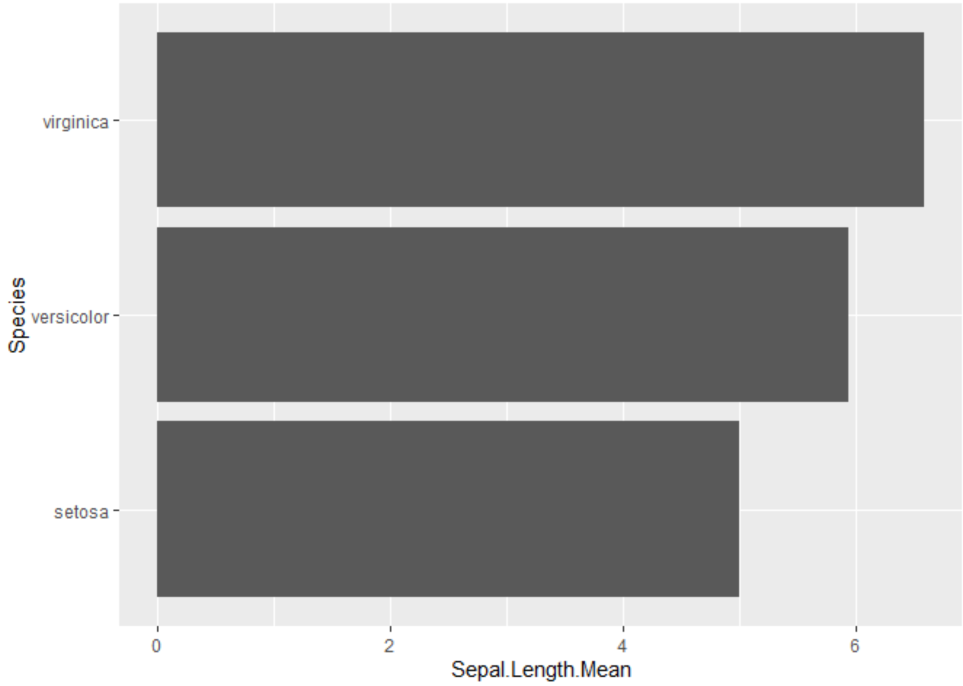
- x and y axis
- latitude and longitude
- radius and angle

```
iris %>%
  group_by(Species) %>%
  summarize(Sepal.Length.Mean = mean(Sepal.Length)) %>%
  ggplot(aes(x = Species, y = Sepal.Length.Mean)) +
  geom_bar(stat = "identity")
```

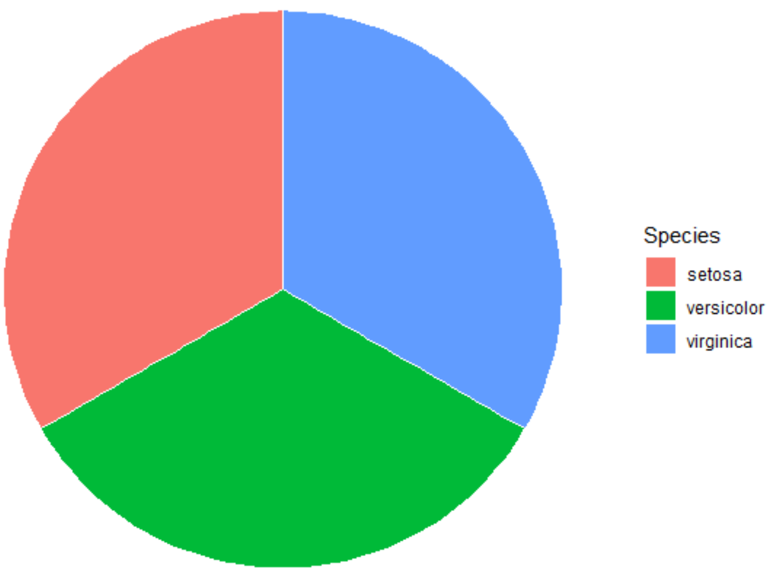



```
# flip the coordinate
iris %>%
```

```
  group_by(Species) %>%
  summarize(Sepal.Length.Mean = mean(Sepal.Length)) %>%
  ggplot(aes(x = Species, y = Sepal.Length.Mean)) +
  geom_bar(stat = "identity") +
  coord_flip()
```



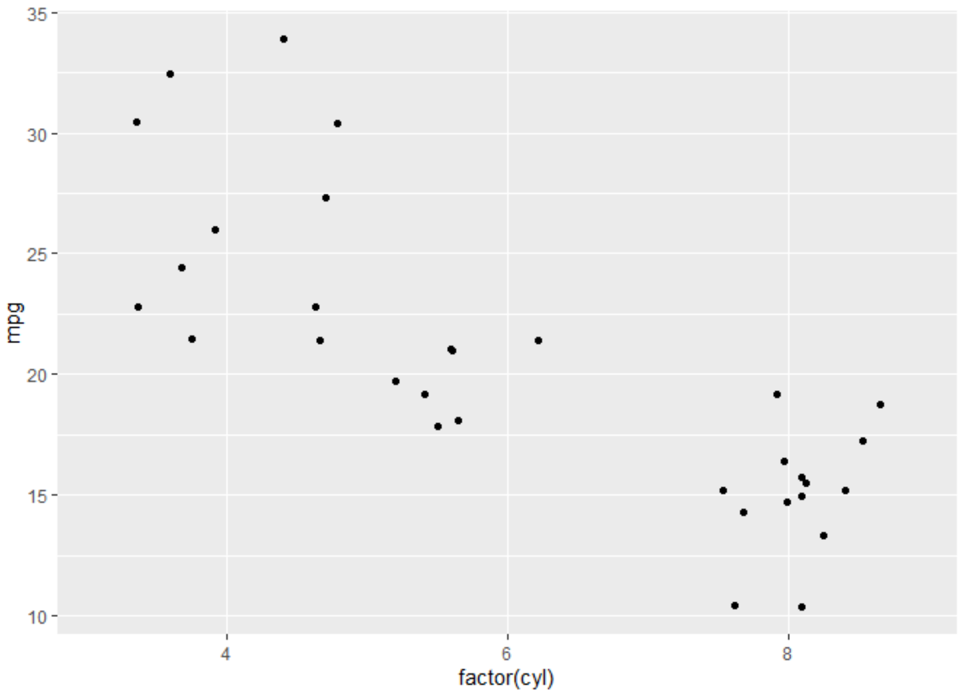

```
# Pie chart by using polar coordinate
iris %>%
  count(Species) %>%
  ggplot(aes(x = "", y = n, fill = Species)) +
  geom_bar(stat = "identity", width = 1, color = "white") +
  coord_polar("y", start = 0) +
  theme_void()
```



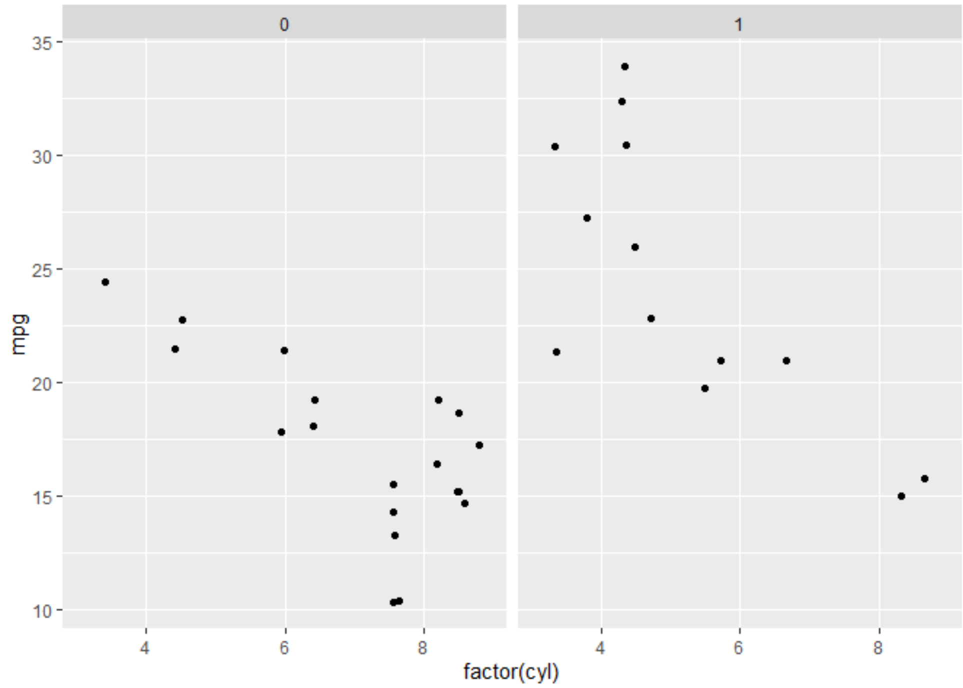
Mapping components: facet

Faceting is generally used to create the same plot for different subsets of the database. Here, we are interested in the distribution of `mpg` under different numbers of `cyl` for automatic (`am=0`) and manual (`am=1`) transmission, respectively.


```
# Jitter plot
mtcars %>%
  ggplot(aes(x = factor(cyl), y = mpg)) +
  geom_point(position = "jitter")
```




```
# faceting
mtcars %>%
  ggplot(aes(x = factor(cyl), y = mpg)) +
  geom_point(position = "jitter") +
  facet_wrap(~am)
```

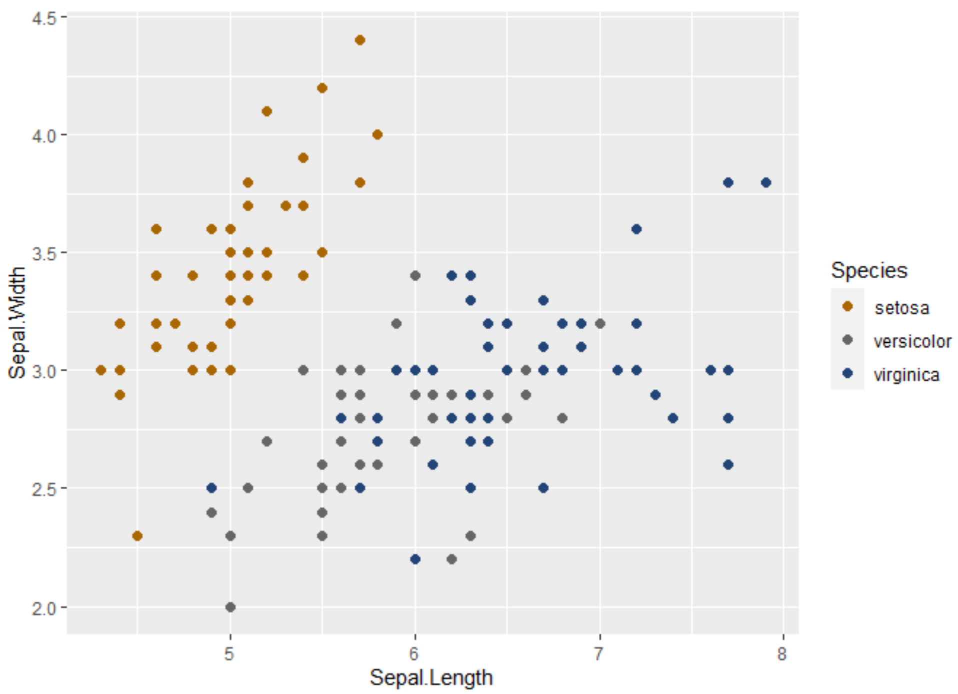


We split the first plot into two facets, one for automatic cars, another for manual one.

Mapping components: theme

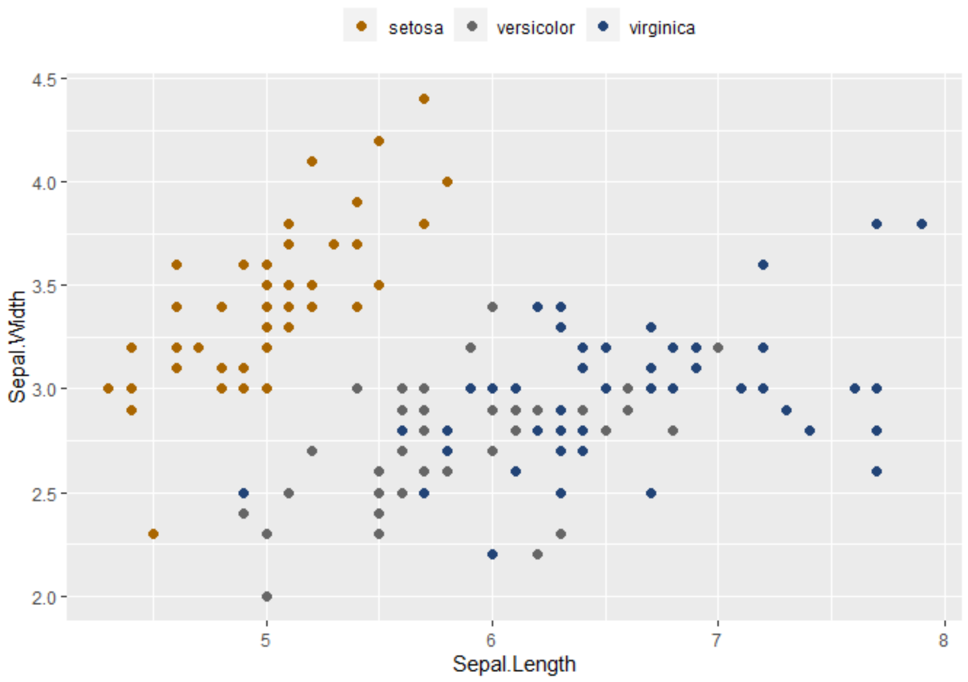
Theme controls the finer points of display like the font size and background color properties. We can not only adjust individual pieces of the plot, including font size, grid lines, legend position, etc, but also adopt a completely custom theme.

```
# Previous scatter plot
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species), size = 2) +
  scale_color_manual(values = c("#aa6600","#666666","#224477"))
```

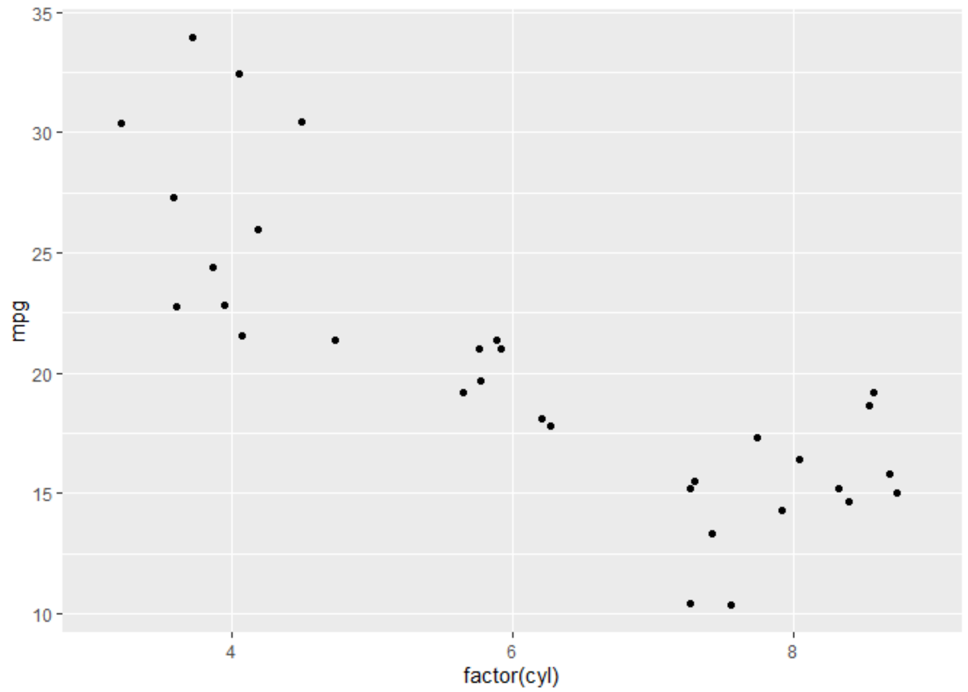



```
# Change the position of legend and remove legend title
iris %>%
  ggplot(aes(x = Sepal.Length, y = Sepal.Width)) +
  geom_point(aes(color = Species), size = 2) +
```

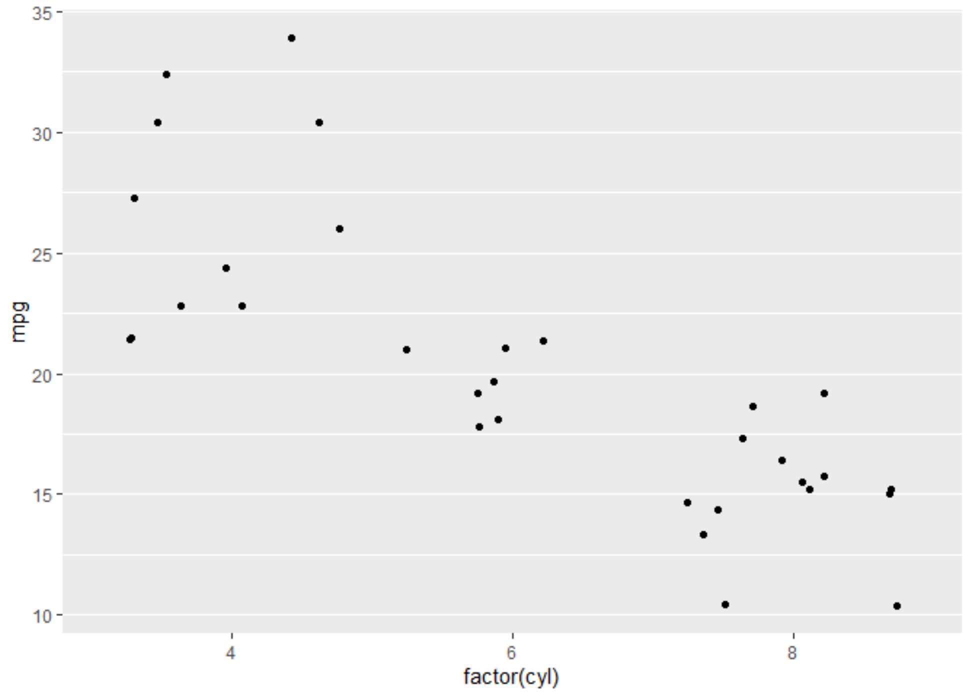
```
  scale_color_manual(values = c("#aa6600","#666666","#224477")) +
  theme(legend.position = "top", legend.title = element_blank())
```




```
# Previous jitter plot
mtcars %>%
  ggplot(aes(x = factor(cyl), y = mpg)) +
  geom_point(position = "jitter")
```




```
# Remove vertical grid lines
mtcars %>%
  ggplot(aes(x = factor(cyl), y = mpg)) +
  geom_point(position = "jitter") +
  theme(panel.grid.major.x = element_blank())
```



The uasge of `ggplot2` is very abundant and flexible. Here, we only introduce the most basic part. In some universities, there will be a separate course on how to use `ggplot2` (say, statistical graphics). If you are interested, please find more details here.

Summary

`tidyverse` is a collection of packages for common data science tasks

- Tidyverse functionality is greatly enhanced using pipes (`%>%` operator)
- Pipes allow you to string together commands to get a flow of results

`dplyr` is a package for data wrangling, with several key verbs (functions)

- `filter()`: subset rows based on a condition
- `group_by()`: define groups of rows according to a condition
- `summarize()`: apply computations across groups of rows
- `arrange()`: order rows by value of a column
- `select()`: pick out given columns
- `mutate()`: create new columns
- `mutate_at()`: apply a function to given columns

`tidyr` is a package for manipulating the structure of data frames

- `pivot_longer()`: make "wide" data longer
- `pivot_wider()`: make "long" data wider

`ggplot2` graphics is comprised of data, layer, scale, coordinate, facet and theme, which are combined by `+`

- layer: `ggplot` for the coordinate system; `geom_point` for scatter or jitter plots; `geom_line`, `geom_bar`, `geom_boxplot`, `geom_histogram` for line charts, bar plots, boxplots and histogram, respectively
- scale: allow changing the color, size and type of points, lines, axis and legend
- coordinate: set a polar coordinate for pie chart `coord_polar`; flip Cartesian coordinates `coord_flip`
- facet: create the same plot for different subsets
- theme: adjust display finer, like font size, background color, or no background layer (`theme_void()`)