IE6600 Computation and Visualization for Analytics

R functions and the grammar of graphics

(updated: 2022-05-16)

R functions

Function

R has a large collection of built-in functions that are called like this:

```
function_name(arg1 = val1, arg2 = val2, ...)
```

Let's try using sum(), which makes regular *sum*mary of numbers. Type su and hit Tab. A pop-up shows you possible completions. Specify sum() by typing more (a "m") to disambiguate, or by using ↑/↓ arrows to select.

```
sum(1,2)
```

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

How to create function

The default syntax for creating a function as follow:

```
variable <- function(arg1, arg2,...){
  your expression/algorithm
}
```

Hello, World!

If we did not include a "Hello, World!" this would not be a serious and sleepy programming class.

```
hello <- function(){
  print("Hello, World!")
}

hello()
```

```
## [1] "Hello, World!"
```

Name masking

You should have no problem predicting the output.

```
f <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
f()
```

```
## [1] 1 2
```

```
rm(f)
```

Name masking (cont'd)

If a name isn't defined inside a function, R will look one level up.

```
x <- 2
g <- function() {
  y <- 1
  c(x, y)
}
g()
```

```
## [1] 2 1
```

```
rm(g)
```

Functions vs variables

If you are using a name in a context where it's obvious that you want a function (e.g.,f(3)), R will ignore objects that are not functions while it is searching.

```
n <- function(x)
  x / 2
o <- function() {
  n <- 10
  n(n)
}
o()
rm(n, o)
```

Exercise

What does the following function return? Make a prediction before running the code yourself.

```
f <- function(x) {
  f <- function(x) {
    f <- function(x) {
      x ^ 2
    }
    f(x) + 1
  }
  f(x) * 2
}
f(10)
```

Every operation is a function call

“To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.”

-John Chambers

Every operation is a function call (cont'd)

```
x <- 1
y <- 2
x+y
```

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

```
`+`(x,y)
```

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

Every operation is a function call (cont'd)

```
for (i in 1:2) print(i)
```

```
## [1] 1
## [1] 2
```

```
`for`(i, 1:2, print(i))
```

```
## [1] 1
## [1] 2
```

Anonymous functions

```
function(x)3()
```

```
## function(x)3()
```

With appropriate parenthesis, the function is called:

```
(function(x)3)()
```

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

Return Values

Functions are generally used for computing some value, so they need a mechanism to supply that value back to the caller.

```
# first build it without an explicit return
double.num <- function(x) {
  x * 2
}
double.num(5)
```

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

```
# now build it with an explicit return
double.num <- function(x) {
  return(x * 2)
}
double.num(5)
```

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

Return Values (cont'd)

```
double.num <- function(x) {
  x * 3
  print("hello")
  x * 2
}
double.num(5)
```

```
## [1] "hello"

## [1] 10
```

```
double.num <- function(x) {
  return(x * 2)
  print("hello")
  return(3)
}
double.num(5)
```

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

Control Statments

Control statements allow us to control the flow of our programming and cause different things to happen, depending on the values of tests. The main control statements are `if` , `else`, `ifelse` and `switch`.

Create one function with if and else

Let's try to create a function to demonstrate if the variable is equal to 1

```
if.one <- function(x){
  if(x==1){
    print("True")
  }else{
    print("False")
  }
}
if.one(1)
if.one(2)
```

Create one function with ifelse

Let's try to create a function to demonstrate if the variable is equal to 1

```
if.one <- function(x){
  ifelse(x==1, "TRUE","FALSE")
}
if.one(1)
if.one(2)
```

Create one function with switch

If we have multiple cases to check, writing else if repeatedly can be cumbersome and inefficient. This is where switch is most useful.

```
multipleCases <- function(x){
  switch(x,
         a="first",
         b="second",
         z="last",
         c="third",
         d="other")
}

multipleCases("a")
```

```
## [1] "first"
```

There is a special argument called ... . This argument will match any arguments not otherwise matched, and can be easily passed on to other functions.

Example

```
f <- function(a,b,c) {
  data.frame(a,b,c)
}
f(a = 1, b = 2, c = 3)
```

```
##   a b c
## 1 1 2 3
```

Example (cont'd)

```
f <- function(...) {
  data.frame(...)
}
f(a = 1, b = 2, c = 3, d = 5, e = 7)
```

```
##   a b c d e
## 1 1 2 3 5 7
```

Tips

Try press Alt-Shift-K.

for Loops

The most commonly used loop is the for loop.

```
for(i in 1:3){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

for Loops (cont'd)

```
# build a vector holding fruit names
fruit <- c("apple", "banana", "pomegranate")
# make a variable to hold their lengths, with all NA to start
fruitLength <- rep(NA, length(fruit))
fruitLength
```

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

```
# give it names
names(fruitLength) <- fruit
fruitLength
```

```
##       apple      banana pomegranate
##          NA          NA          NA
```

for Loops (cont'd)

```
for (i in fruit){
  fruitLength[i] <- nchar(i)
}
fruitLength
```

```
##       apple      banana pomegranate
##           5           6          11
```

Apply Family

Built into R is the `apply` function and all of its common relatives such as `lapply`, `sapply` and `mapply`. Each has its quirks and necessities and is best used in different situations.

apply

`apply` is the first member of this family that users usually learn, and it is also the most restrictive. It must be used on a matrix, meaning all of the elements must be of the same type whether they are character, numeric or logical.

```
theMatrix <- matrix(1:9, nrow=3)
# sum the rows
apply(theMatrix, 1, sum)
```

```
## [1] 12 15 18
```

```
# sum the columns
apply(theMatrix, 2, sum)
```

```
## [1]  6 15 24
```

apply (cont'd)

Notice that this could alternatively be accomplished using the built-in rowSums and colSums functions, yielding the same results.

```
rowSums(theMatrix)
```

```
## [1] 12 15 18
```

```
colSums(theMatrix)
```

```
## [1]  6 15 24
```

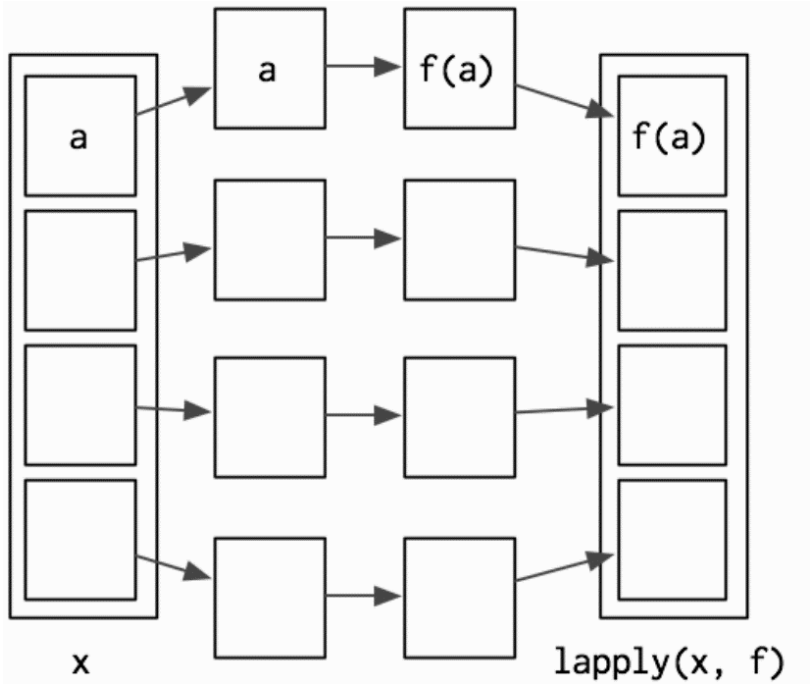
lapply and sapply

Basic grammar

```
lapply(x, FUN, ...)
sapply(x, FUN, ...)
```

lapply

lapply works by applying a function to each element of a list and returning the results as a list.



lapply

lapply works by applying a function to each element of a list and returning the results as a list.

```
theList <- list(A=1:3, B=1:5, C=-1:1, D=2)
lapply(theList, sum)
```

```
## $A
## [1] 6
##
## $B
## [1] 15
##
## $C
## [1] 0
##
## $D
## [1] 2
```

sapply

sapply is a user-friendly version and wrapper of lapply by default returning a vector.

```
theList <- list(A=1:3, B=1:5, C=-1:1, D=2)
sapply(theList, sum)
```

```
##  A  B  C  D
##  6 15  0  2
```

mapply

Perhaps the most-overlooked-when-so-useful member of the apply family is mapply, which applies a function to each element of multiple lists.

```
firstList <-
  list(A = matrix(1:16, 4),
       B = matrix(1:16, 2),
       C = data.frame(1:5))
secondList <-
  list(A = matrix(1:16, 4),
       B = matrix(1:16, 8),
       C = data.frame(15:1))
# test element-by-element if they are identical
mapply(identical, firstList, secondList)
```

```
##     A     B     C
##  TRUE FALSE FALSE
```

mapply (cont'd)

Lets create one small function with mapply

```
simpleFunc <- function(x, y) {
  nrow(x) + nrow(y)
}
mapply(simpleFunc, firstList, secondList)
```

```
##  A  B  C
##  8 10 20
```

The grammar of graphics

Common plots

Common statistical plots:

- Bar chart
- Scatter plot
- Line chart
- Box plot
- Histogram

Common plots (cont'd)

Theses can always be changed:

Scatter plot

- continuous varialbe vs continuous variable

Line plot

- continuous varialbe vs continuous variable

Box plot

- categorical varialbe vs continuous variable

Common plots (cont'd)

Histogram

- continuous variable vs continuous variable

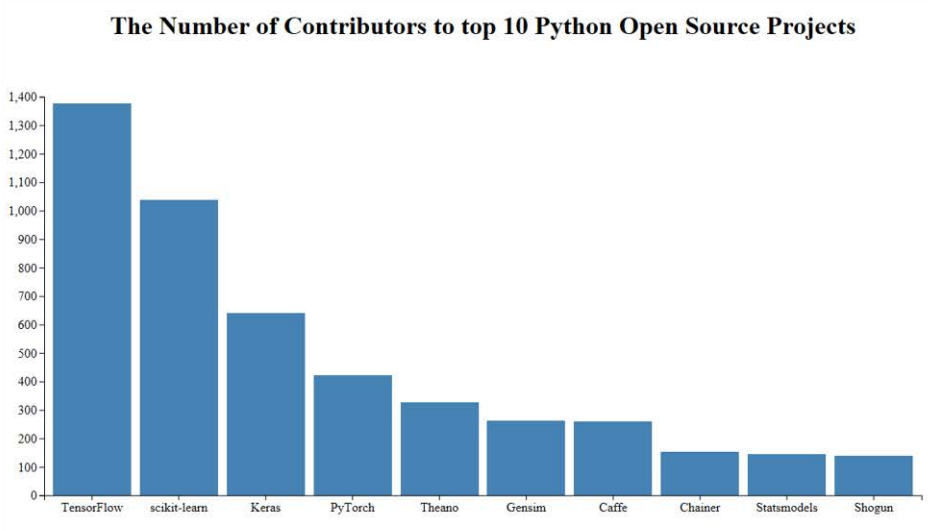
Bar plot

- categorical varialbe vs continuous variable

A basic plot

Title of this graph
A description of the data or something worth highlighting to set stage

Scale
Increments that make sense can increase readability, as well as shift focus



Coordinate System
You map data differently with a scatterplot than you do with a pie chart. It's x- and y-coordinates in one and angles with the others.

Context
If your audience is unfamiliar with the data, it's your job to clarify what values represent and explain how people should read your visualization

Visual Cues
Visualization involves encoding data with shapes, colors and sizes. Which cues you choose depends on your data and your goals

A basic plot (cont'd)

Let's draw a scatterplot of A vs C

Table 1. Simple dataset.

A	*B*	*C*	*D*
2	3	4	a
1	2	1	a
4	5	15	b
9	10	80	b

[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

A basic plot (cont'd)

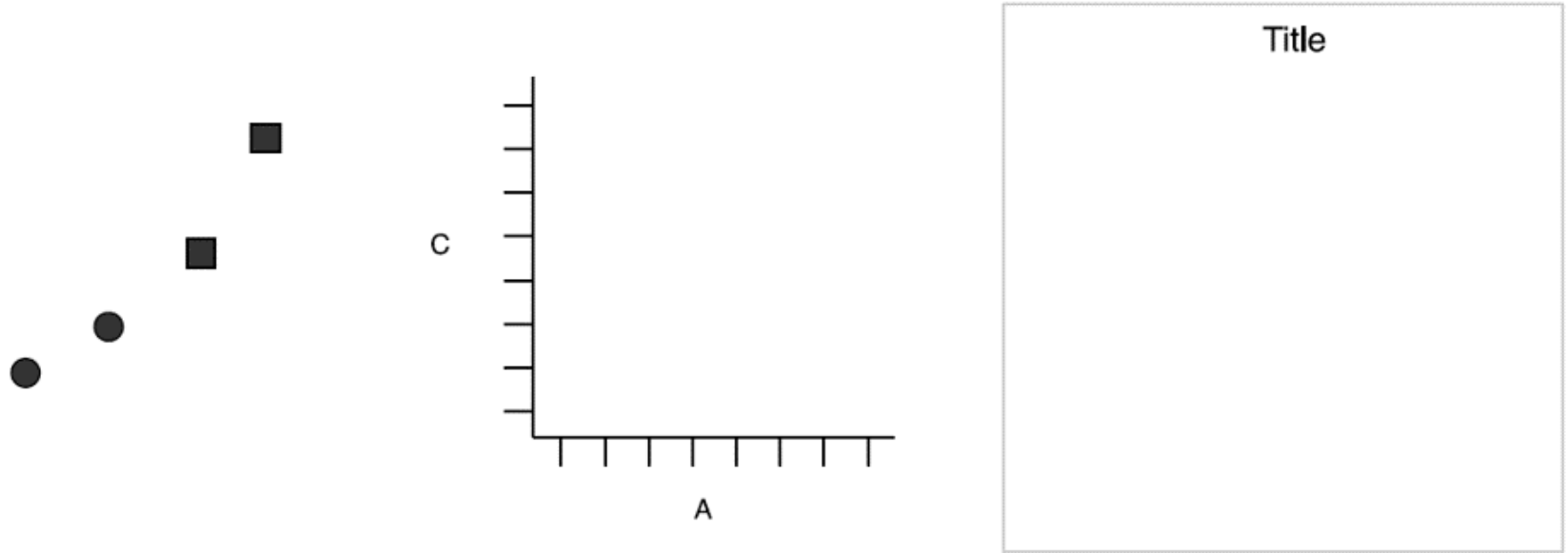
Mapping A to x-position, C to y-position, and D to shape

x	y	Shape
2	4	circle
1	1	circle
4	15	square
9	80	square

[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

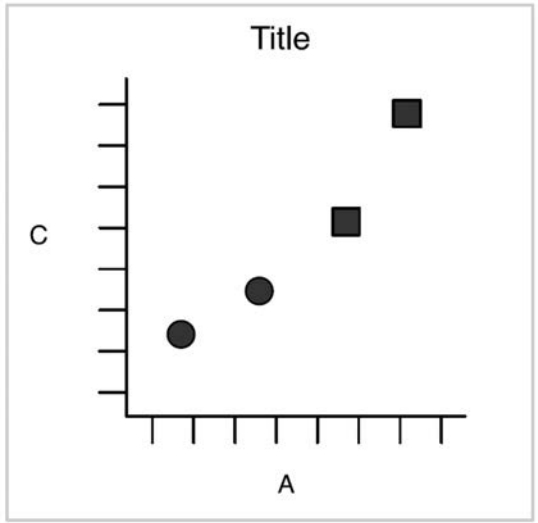
A basic plot (cont'd)

- Geometric objects
- scales
- coordinate system (From left to right)



A basic plot (cont'd)

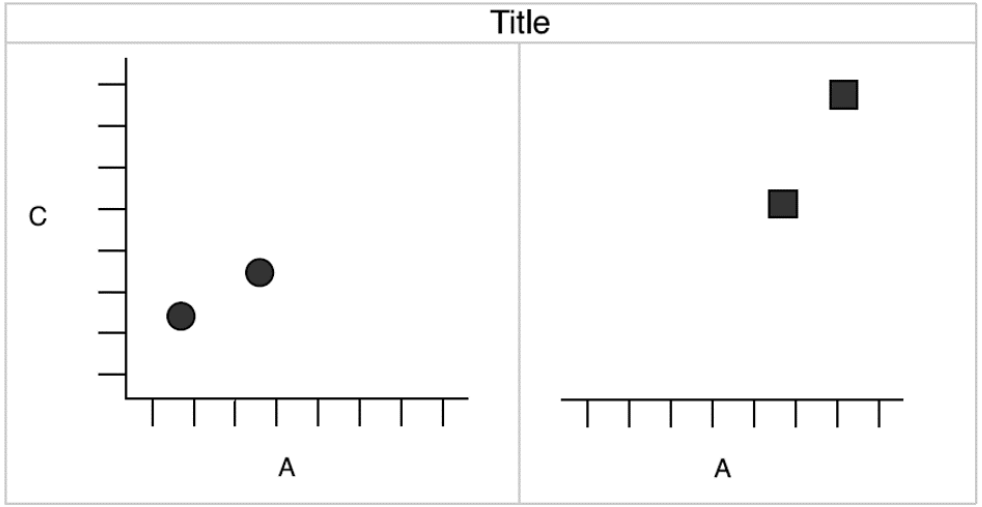
Then we have one



[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

A more complicated plot

We created a plot by faceting



[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

Components of the plots

- Layers:
  - Dataset
  - Aesthetic mapping (color, shape, size, etc.)
  - Statistical transformation
  - Geometric object (line, bar, dots, etc.)
  - Position adjustment
- Scale (optional)
- Coordinate system
- Faceting (optional)
- Defaults

[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

ggplot2 full syntax

```
ggplot(data = <DATASET>,
       mapping = aes( <MAPPINGS>) +
        layer(geom = <GEOM>,
              stat = <STAT>,
              position = <POSITION>) +
        <SCALE_FUNCTION>() +
        <COORDINATE_FUNCTION>() +
        <FACET_FUNCTION>()
```

Previous example

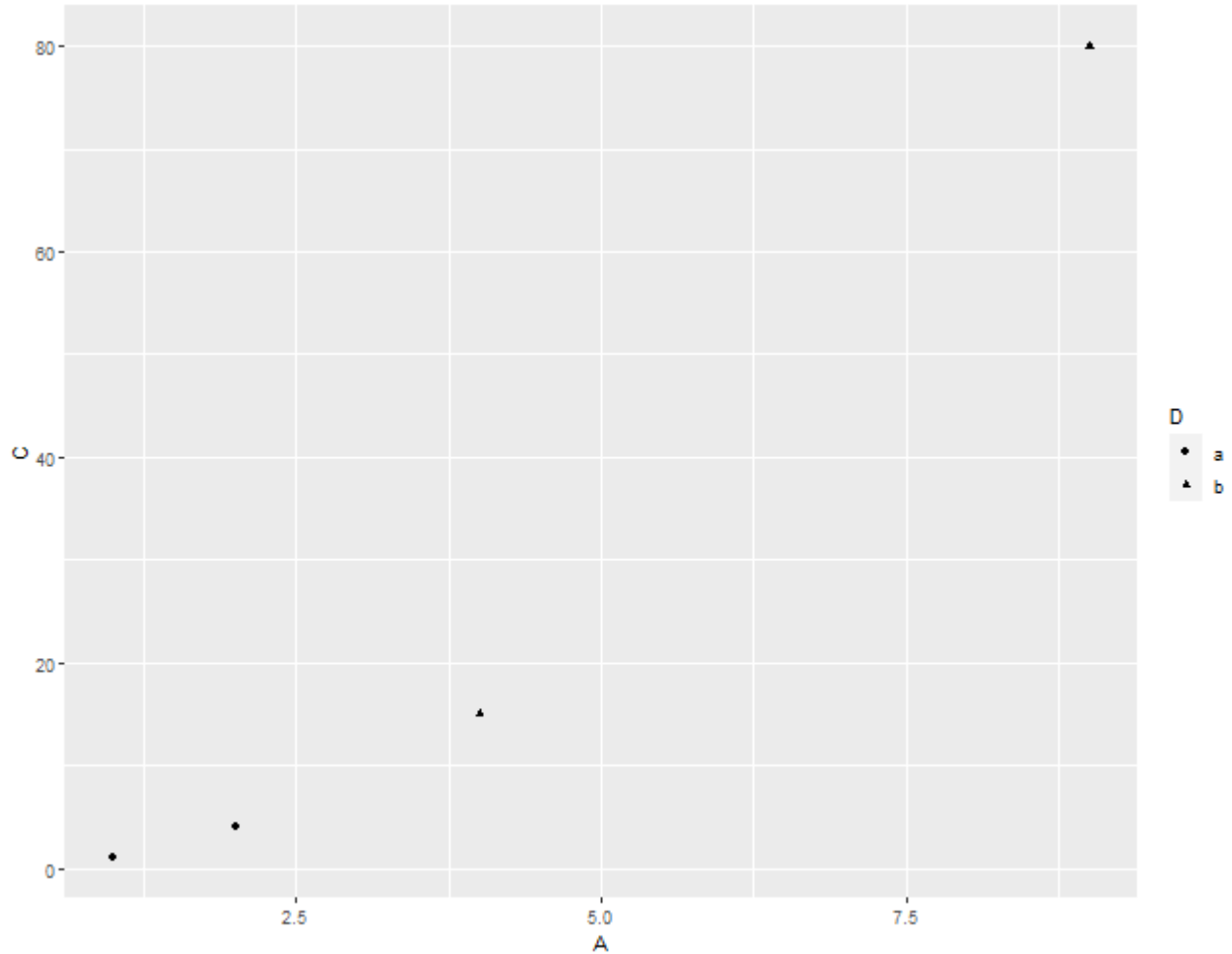
```
df <- data.frame(
  A = c(2, 1, 4, 9),
  B = c(3, 2, 5, 10),
  C = c(4, 1, 15, 80),
  D = c("a", "a", "b", "b")
)
df
```

```
##   A  B  C D
## 1 2  3  4 a
## 2 1  2  1 a
## 3 4  5 15 b
## 4 9 10 80 b
```

Thoughts?

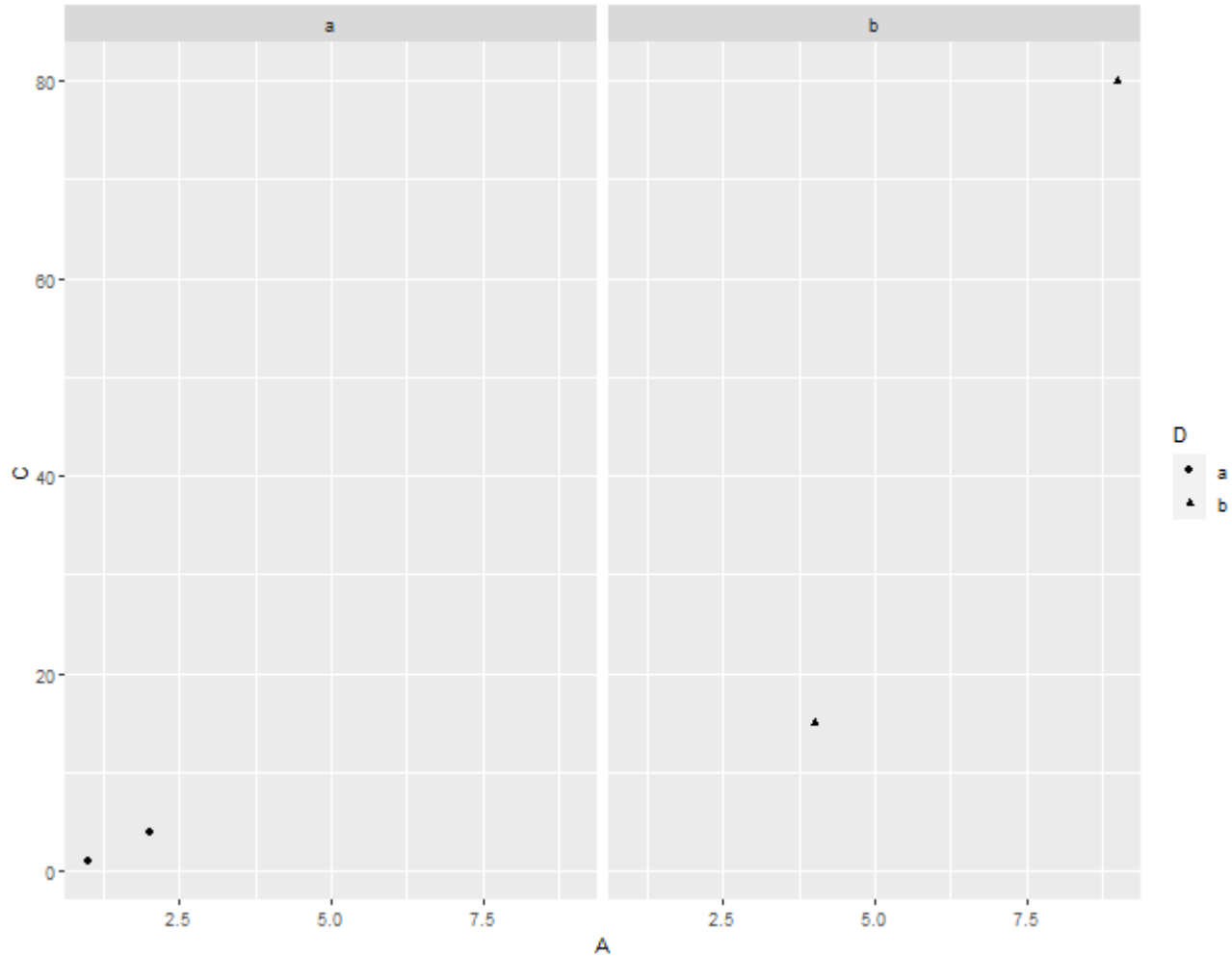
- What kind of layers?
  - Dataset?
  - Aesthetic mapping?
  - Statistical transformation?
  - Geometric object?
  - Position adjustment?
- Scale?
- Coordinate system?
- Faceting?

```
ggplot(data = df,
       mapping = aes(x = A, y = C, shape = D)) +
  layer(geom = "point",
        stat = "identity",
        position = "identity") +
  scale_x_continuous() +
  scale_y_continuous() +
  coord_cartesian() +
  facet_null()
```



Facetting with grid

```
ggplot(data = df,
       mapping = aes(x = A, y = C, shape = D)) +
  layer(geom = "point",
        stat = "identity",
        position = "identity") +
  scale_x_continuous() +
  scale_y_continuous() +
  coord_cartesian() +
  facet_grid( ~ D)
```



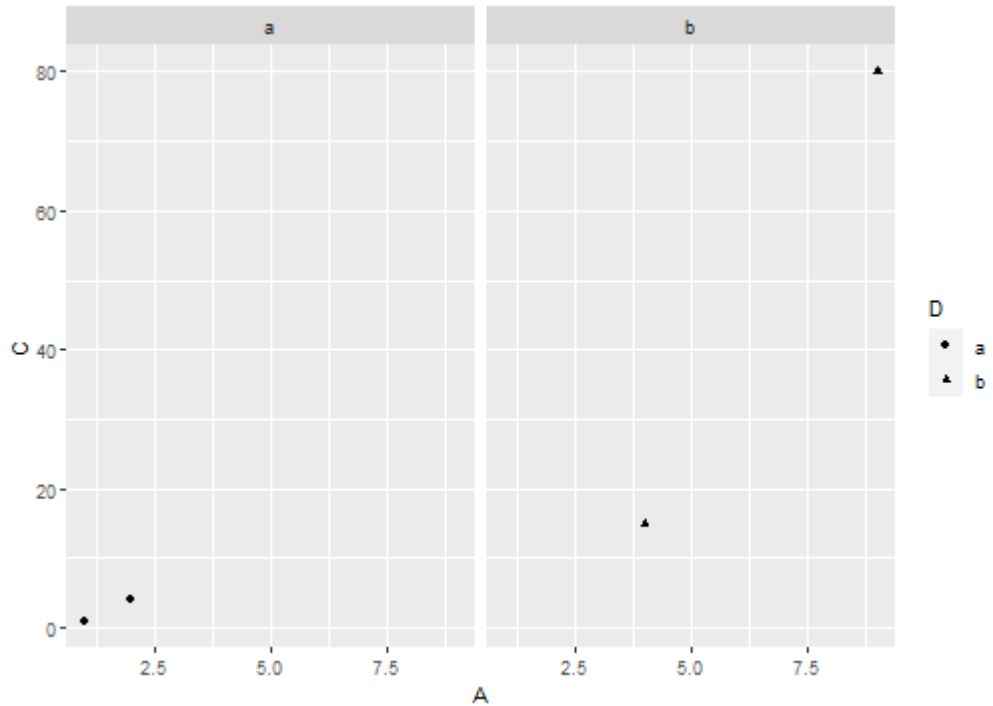
A typical graph template

```
ggplot(data = <DATASET> ,
       mapping = aes(<MAPPINGS)) +
<GEOM_FUNCTION>()
```

[1] Wickham, Hadley, and Garrett Grolemund. R For Data Science. OReilly, 2017.

With geom_ function

```
ggplot(data = df, mapping = aes(x = A, y = C, shape = D)) +
  geom_point()+
  facet_grid( ~ D)
```



Dataset - Fuel economy in cars

```
library(tidyverse)
```

```
head(mpg,5)
```

```
## # A tibble: 5 x 11
##   manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
##   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa~
## 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
## 3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa~
## 4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa~
## 5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa~
```

Dataset - Interviewing data

```
mpg[,c("displ","hwy")]
```

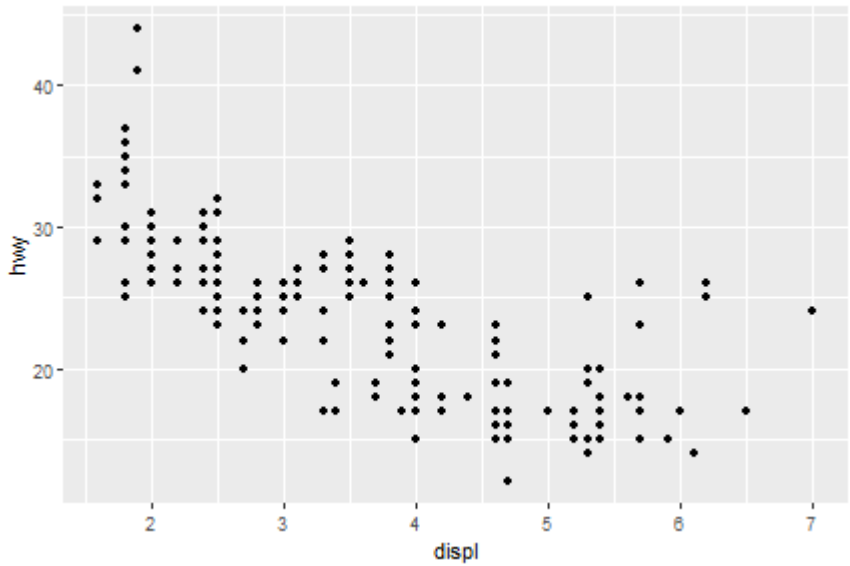
```
## # A tibble: 234 x 2
##    displ   hwy
##    <dbl> <int>
##  1   1.8    29
##  2   1.8    29
##  3   2      31
##  4   2      30
##  5   2.8    26
##  6   2.8    26
##  7   3.1    27
##  8   1.8    26
##  9   1.8    25
## 10   2      28
## # ... with 224 more rows
```

Dataset - Creating base

```
ggplot(data = mpg)
```

Dataset - Creating plot

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```



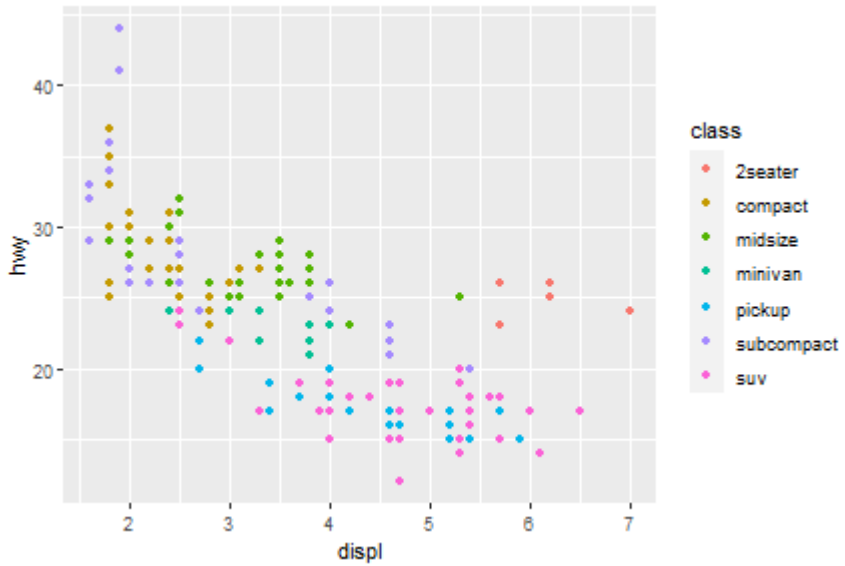
Exercise

- How many rows are in mtcars? How many columns?
- What does the drv variable describe? Read the help for ?mpg to find out.
- Make a scatterplot of hwy versus cyl.

Aesthetic mappings

We can change levels of *size*, *shape*, *color*, *fill*, *alpha* etc. inside of *aes()*

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



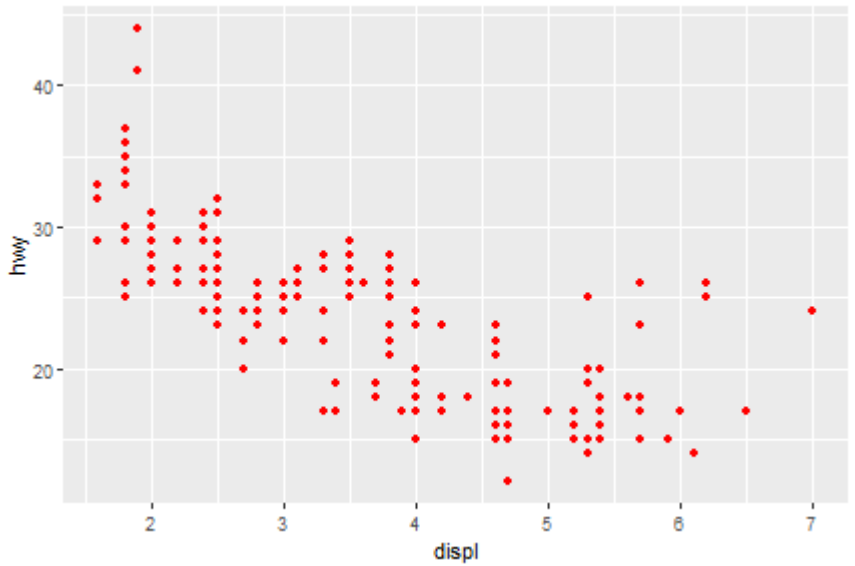
colour instead of color

Well, if you prefer British English, you can use

Aesthetic mappings - outside of aes()

You can also set the aesthetic properties of your geom manually.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "red")
```

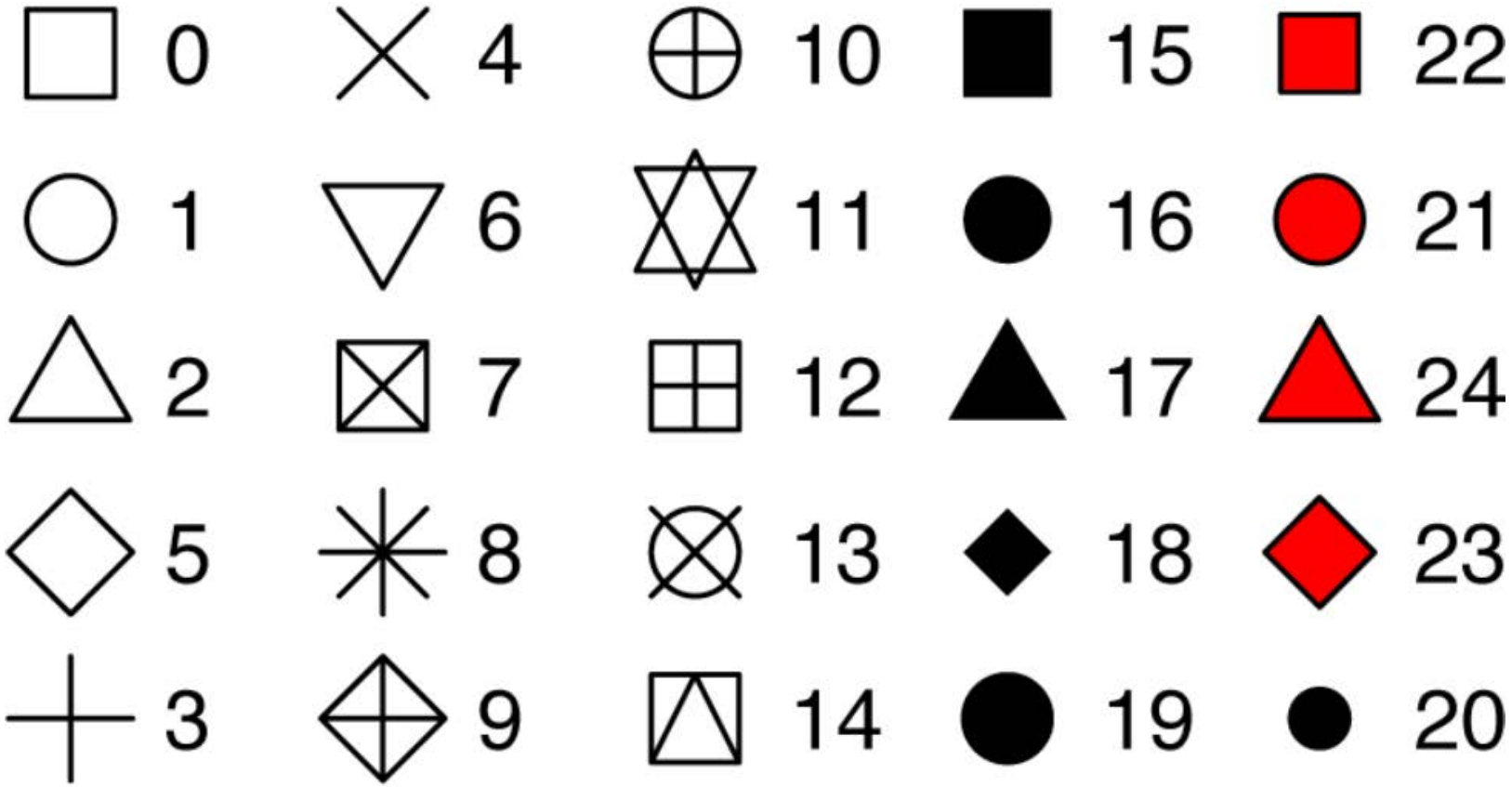


Here, the color doesn't convey information about a variable, but only changes the appearance of the plot.

Aesthetic mappings - outside of aes() (cont'd)

- The name of a color as a character string
- The size of a point in mm
- The shape of a point as a number

25 built-in shapes identified by numbers



geom_() functions

- Geometric object
- Statistical transformation
- Position adjustment

Common geom functions with geometric objects

- *geom_bar*, bar chart
- *geom_histogram*, histogram
- *geom_point*, scatterplot
- *geom_qq*, quantile-quantile plot
- *geom_boxplot*, boxplot
- *geom_line*, line chart

Statistical transformation

Name	Description
bin	Divide continuous range into bins, and count number of points in each
boxplot	Compute statistics necessary for boxplot
contour	Calculate contour lines
density	Compute 1d density estimate
identity	Identity transformation, $f(x) = x$
jitter	Jitter values by adding small random value
qq	Calculate values for quantile-quantile plot
quantile	Quantile regression
smooth	Smoothed conditional mean of y given x
summary	Aggregate values of y for given x
unique	Remove duplicated observations

[1] [Wickham, Hadley. “A Layered Grammar of Graphics.” Journal of Computational and Graphical Statistics, vol. 19, no. 1, 2010, pp. 3–28., doi:10.1198/jcgs.2009.07098.]

Common geom with statistical transformation

Typically, you will create layers using a geom_ function.

- *geom_bar*, bar chart
  - stat="count"
- *geom_histogram*, histogram
  - stat="bin"
- *geom_point*, scatterplot
  - stat="identity"

Common geom with statistical transformation (cont'd)

- *geom_qq*, quantile-quantile plot
  - stat="qq"
- *geom_boxplot*, boxplot
  - stat="boxplot"
- *geom_line*, line chart
  - stat="identity"

The defaults stat and position of geom_

Check the documentation

```
?geom_line
```

Check the function

```
geom_line
```

```
## function (mapping = NULL, data = NULL, stat = "identity", position = "identity",
##     na.rm = FALSE, orientation = NA, show.legend = NA, inherit.aes = TRUE,
##     ...)
## {
##     layer(data = data, mapping = mapping, stat = stat, geom = GeomLine,
##         position = position, show.legend = show.legend, inherit.aes = inherit.aes,
##         params = list(na.rm = na.rm, orientation = orientation,
##             ...))
## }
## <bytecode: 0x0000000024c69df8>
## <environment: namespace:ggplot2>
```

Set of rules

- Use *geom_* function to make variables visible on the screen.
- Use *stat_* function and define geom shape as an argument inside *geom_*.
- Or use *geom_* and define statistical transformation as an argument inside *stat_*.

Common geom with position adjustments

- point: geom_point, geom_jitter

Scale syntax

```
scale_<name>_<prepacked scale>()
```

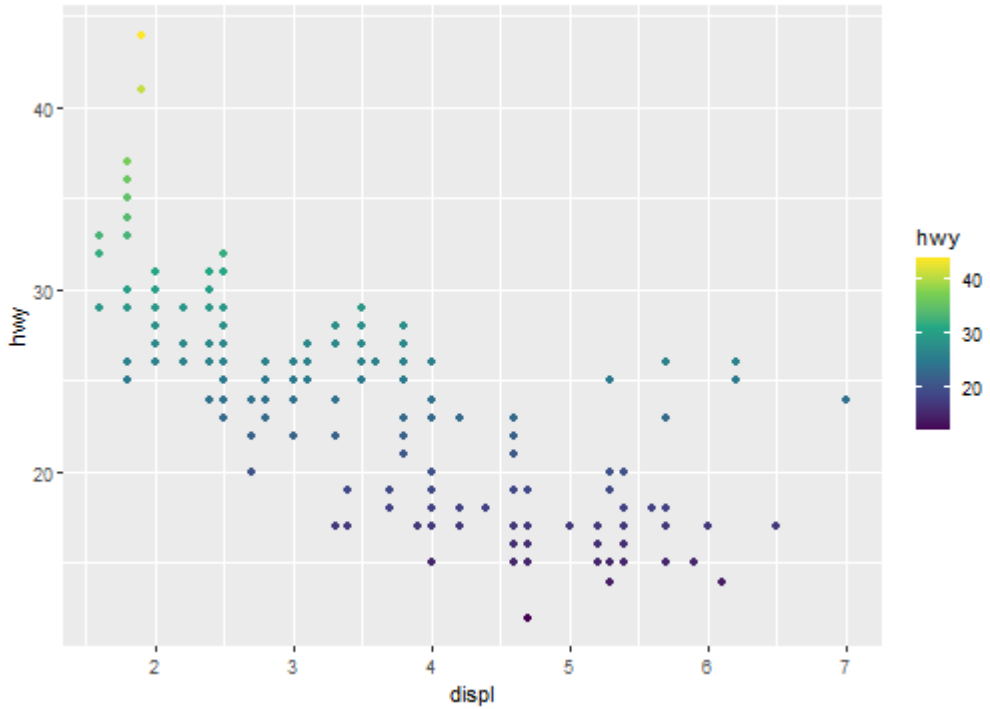
Several common scale functions:

```
labs() xlab() ylab() ggtitle()
lims() xlim() ylim()
scale_colour_brewer()
scale_colour_continuous()
```

[1] ggplot2 references

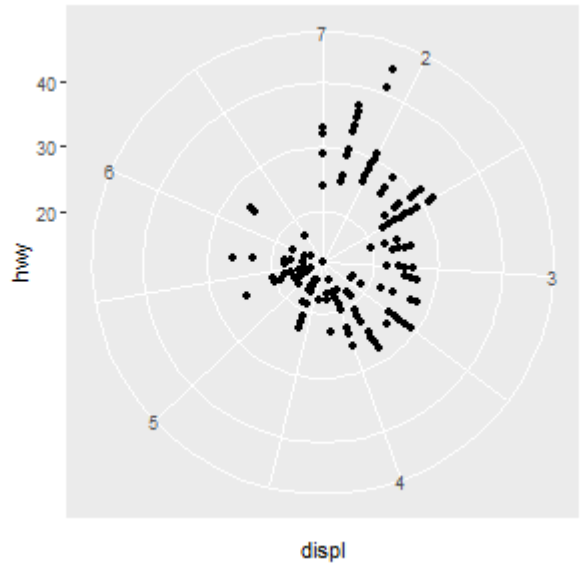
Scale example

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color=hwy))+
  scale_colour_continuous(type="viridis")
```



Coordinate system

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))+
  coord_polar()
```



[1] ggplot2 references

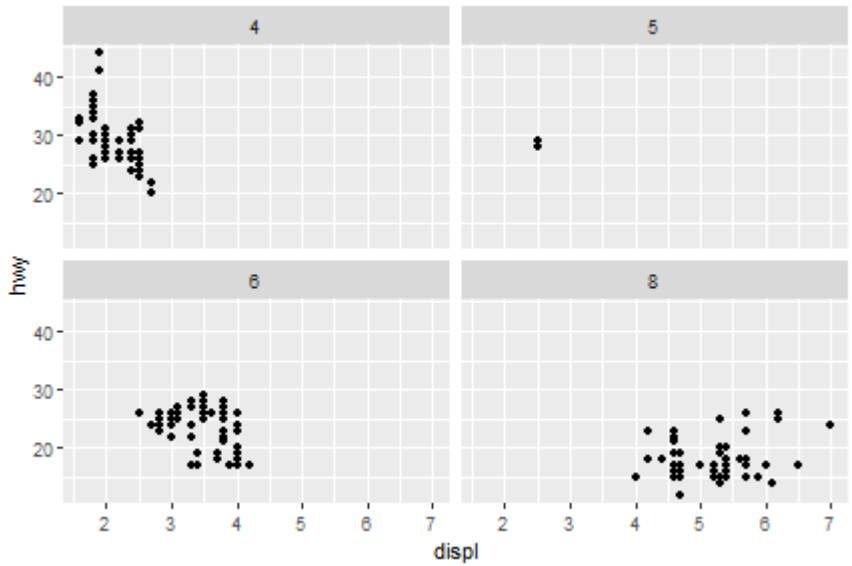
Faceting - grid

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))+
  facet_grid(.~year)
```

[1] ggplot2 references

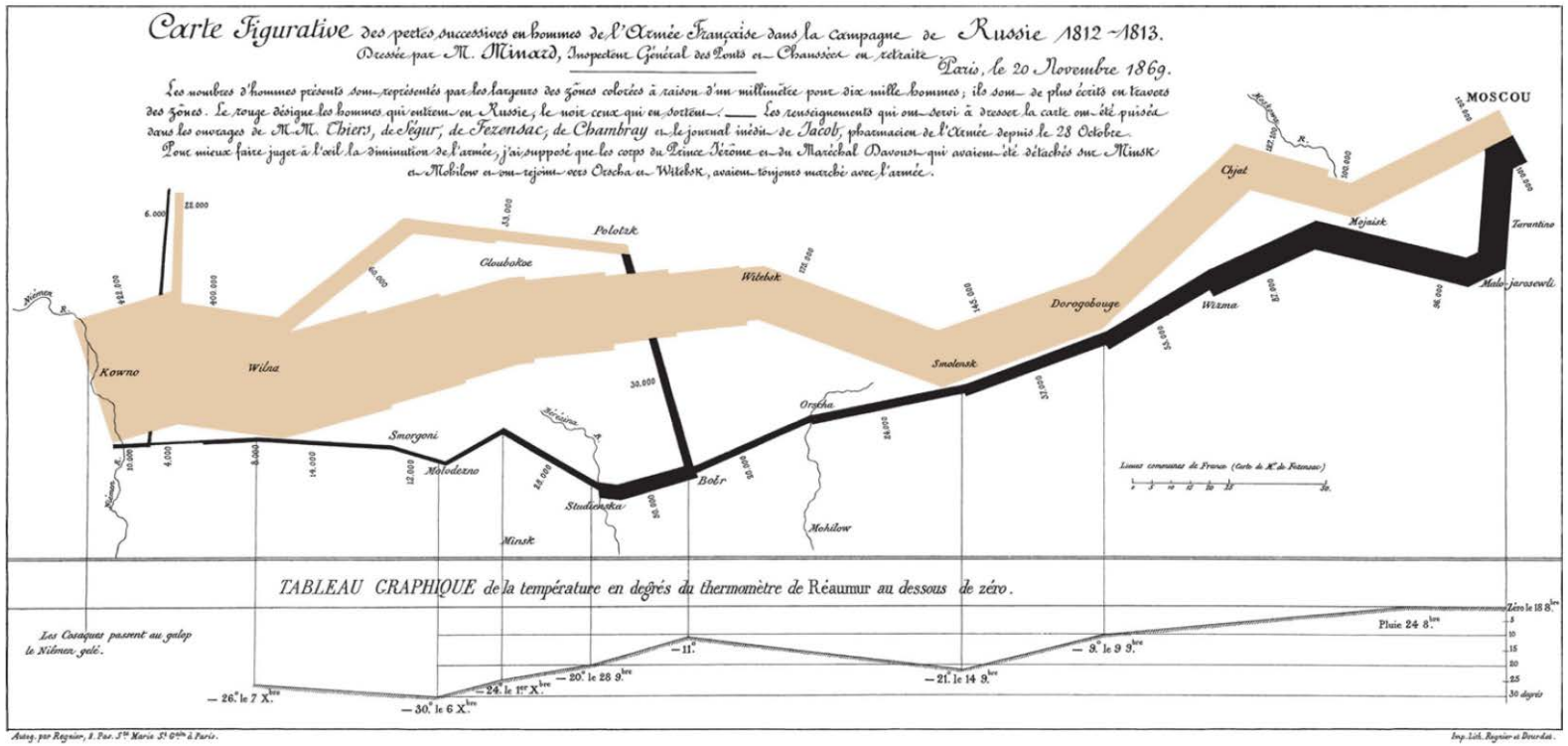
Faceting - wrap

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))+
  facet_wrap(.~cyl)
```



[1] ggplot2 references

A more complicated-embedded grammar



A more complicated-embedded grammar (cont'd)

Let's reproduce the top part of Minard's famous depiction of Napoleon's march on Russia.

This graphic can be thought of as a compound graphic:

- The top part displays the number of troops during the advance and the retreat
- The bottom part shows the temperature during the advance

We will focus on the top part of the graphic. This part displays two datasets: cities and troops. Each city has a position (a latitude and longitude) and a name, and each troop observation has a position, a direction (advance or retreat), and number of survivors.

Load data: minard-troops.txt and minard-cities.txt

```
troops <- read.table("minard-troops.txt", header=T)
cities <- read.table("minard-cities.txt", header=T)
```

troops

```
head(troops)
```

```
##   long  lat survivors direction group
## 1 24.0 54.9    340000         A     1
## 2 24.5 55.0    340000         A     1
## 3 25.5 54.5    340000         A     1
## 4 26.0 54.7    320000         A     1
## 5 27.0 54.8    300000         A     1
## 6 28.0 54.9    280000         A     1
```

cities

```
head(cities)
```

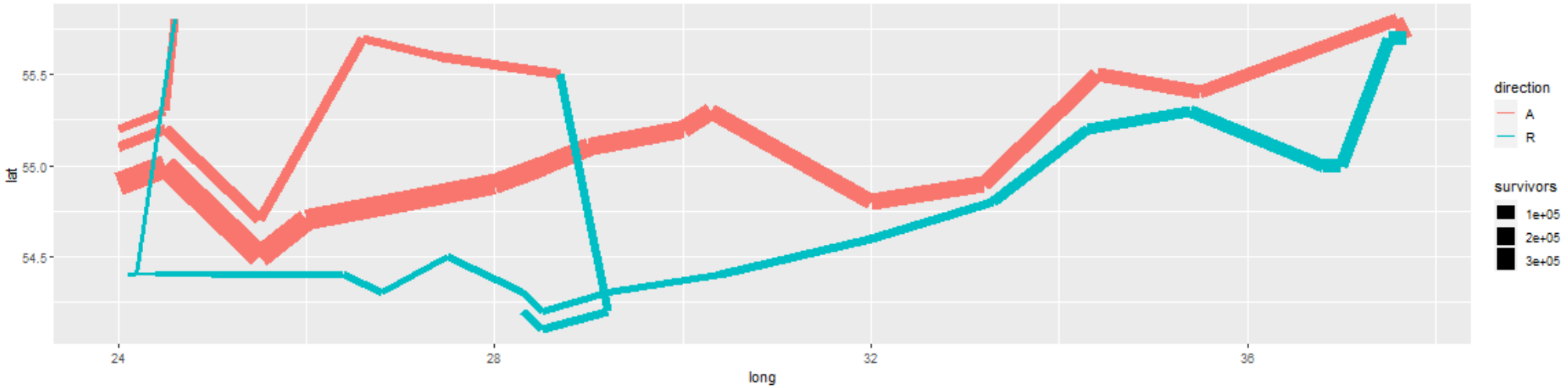
```
##   long  lat      city
## 1 24.0 55.0     Kowno
## 2 25.3 54.7     Wilna
## 3 26.4 54.4  Smorgoni
## 4 26.8 54.3 Moiodexno
## 5 27.7 55.2 Gloubokoe
## 6 27.6 53.9     Minsk
```

How would we create this graphic with the layered grammar?

- Start with the essence of the graphic: a path plot of the troops data, mapping direction to color and number of survivors to size.
- Then take the position of the cities as an additional layer
- Then polish the plot by carefully tweaking the default scales

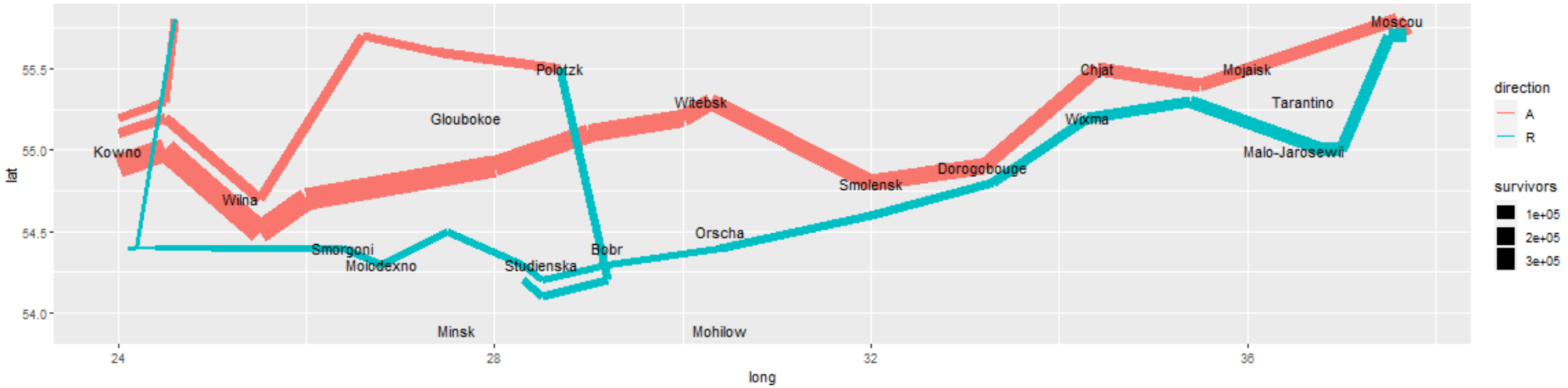
Start with the essence of the graphic: a path plot of the troops data, mapping direction to color and number of survivors to size.

```
plot_troops <- ggplot(troops, aes(long, lat)) +
  geom_path(aes(size = survivors, colour = direction, group = group))
plot_troops
```



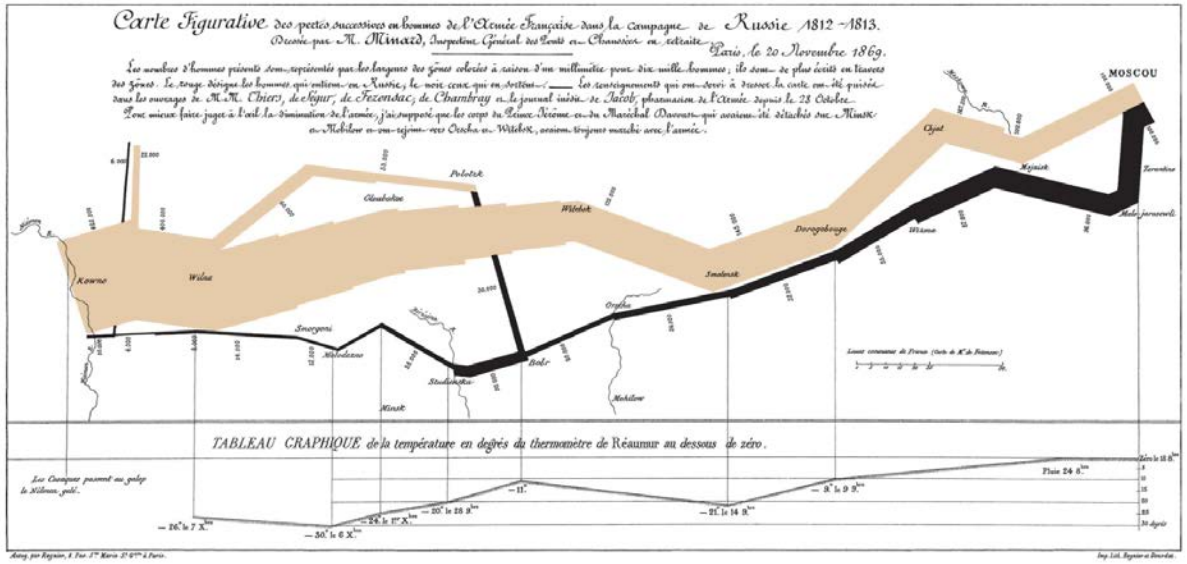
Then take the position of the cities as an additional layer

```
plot_both <- plot_troops +
  geom_text(aes(label = city), size = 4, data = cities)
plot_both
```



Then polish the plot by carefully tweaking the default scales

```
plot_polished <- plot_both +
  scale_size(
    range = c(0, 12),
    # breaks = c(10000, 20000, 30000),
    # labels = c("10,000", "20,000", "30,000")
  ) +
  scale_color_manual(values = c("tan", "grey50")) +
  coord_map() +
  labs(title = "Map of Napoleon's Russian campaign of 1812") +
  theme_void() +
  theme(legend.position = "none")
plot_polished
```



Map of Napoleon's Russian campaign of 1812