Day 2

(Very rough) time plan

Friday Nov 19

09:15-10:00

- Introduction to R and RStudio
- Set up and get going
- Do Exercise 1

10:15 - 12:00

- Go through Exercise 1
- R packages and the Tidyverse
- Rectangular and tidy data
- Working with files
- Exercise 2
- Go through Exercise 2

12:45 - 14:00

- Manipulating data with dplyr
- Exercise 3

14:15 - 16:00

- Go through Exercise 3
- Basic plotting
- Exercise 4
- Go through exercise 4 together

Monday Nov 22

09:15 - 11:30

- Programming basics
  - For loops + Ex 5 (09:15 - 10:30)
  - Ex 5 + If statements + Ex 6 (10:45 - 11:30)
  - Go through exercise 6 (11:30 - 12:00)

12:45

- R scripts
  - Running R on the command line
  - Command line arguments
- Plotting with ggplot2

Iteration

In programming it’s important to **reduce duplication**. A rule of thumb is to *never copy and paste the same code more than twice*.

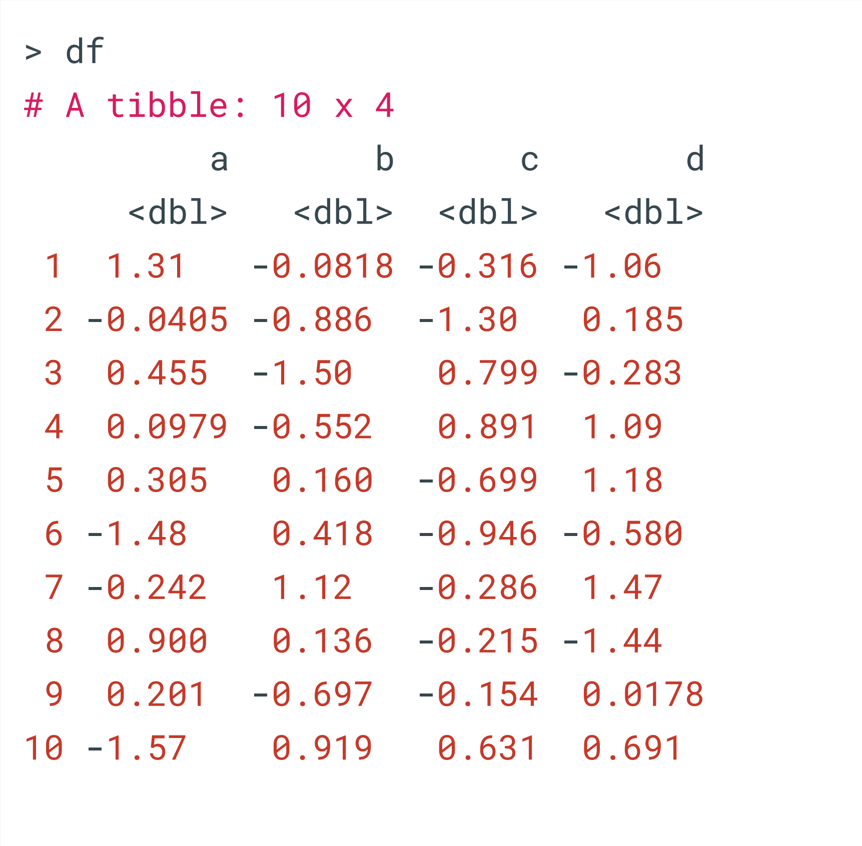
Iteration helps you to *do the same thing to multiple inputs* (e.g. repeating the same operation on different columns, or on different datasets...).

There are a few ways to iterate in R:

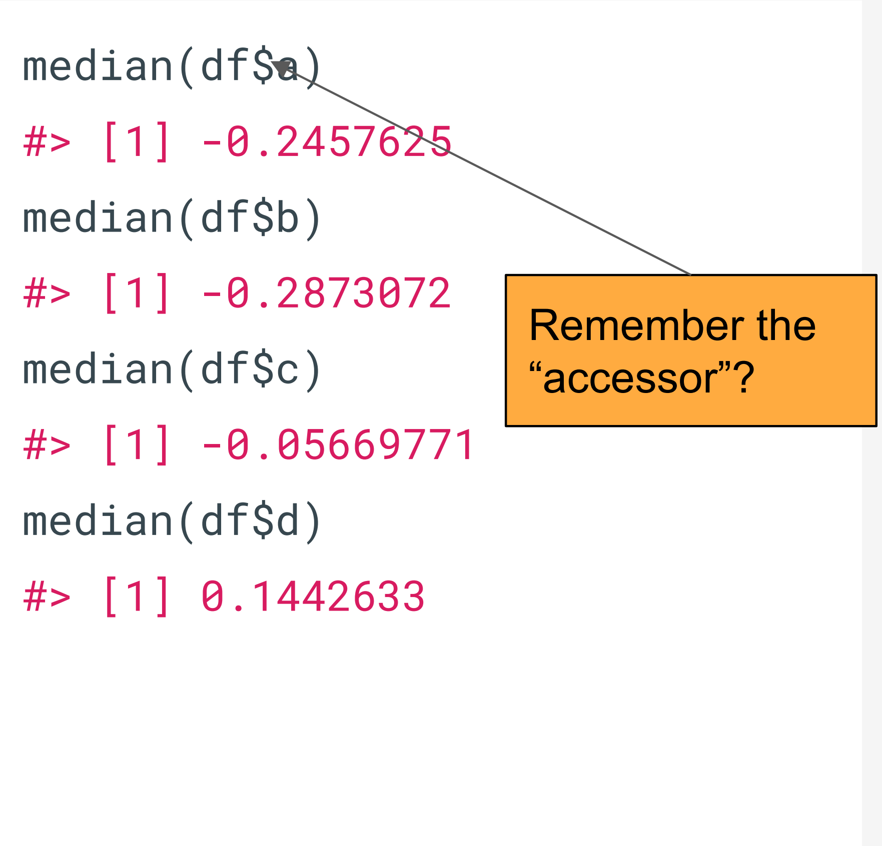
- loops (for loops and while loops - we only focus on the for loop)
- The tidyverse map() functions (even more condense than for loops, but require more knowledge about R than you will get here...)

Iteration

We have this simple data frame and want to compute the median of each column. We can copy and paste the median() function like this:



```
> df
# A tibble: 10 x 4
          a       b      c       d
      <dbl>   <dbl>  <dbl>   <dbl>
 1  1.31   -0.0818 -0.316 -1.06
 2 -0.0405 -0.886  -1.30   0.185
 3  0.455  -1.50    0.799 -0.283
 4  0.0979 -0.552   0.891  1.09
 5  0.305   0.160  -0.699  1.18
 6 -1.48    0.418  -0.946 -0.580
 7 -0.242   1.12   -0.286  1.47
 8  0.900   0.136  -0.215 -1.44
 9  0.201  -0.697  -0.154  0.0178
10 -1.57    0.919   0.631  0.691
```



```
median(df$a)
#> [1] -0.2457625
median(df$b)
#> [1] -0.2873072
median(df$c)
#> [1] -0.05669771
median(df$d)
#> [1] 0.1442633
```

Remember the “accessor”?

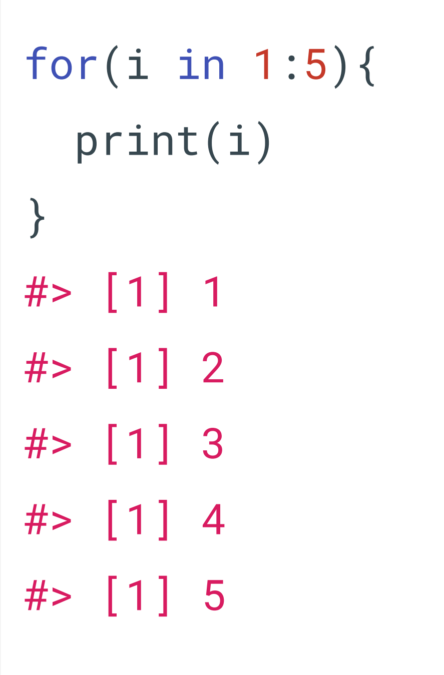
for loop

We have this simple data frame and want to compute the median of each column. We can use a **for loop**:

```
output <- vector("double", ncol(df))  # 1. output
for (i in 1:ncol(df)) {               # 2. sequence
  output[i] <- median(df[[i]])        # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

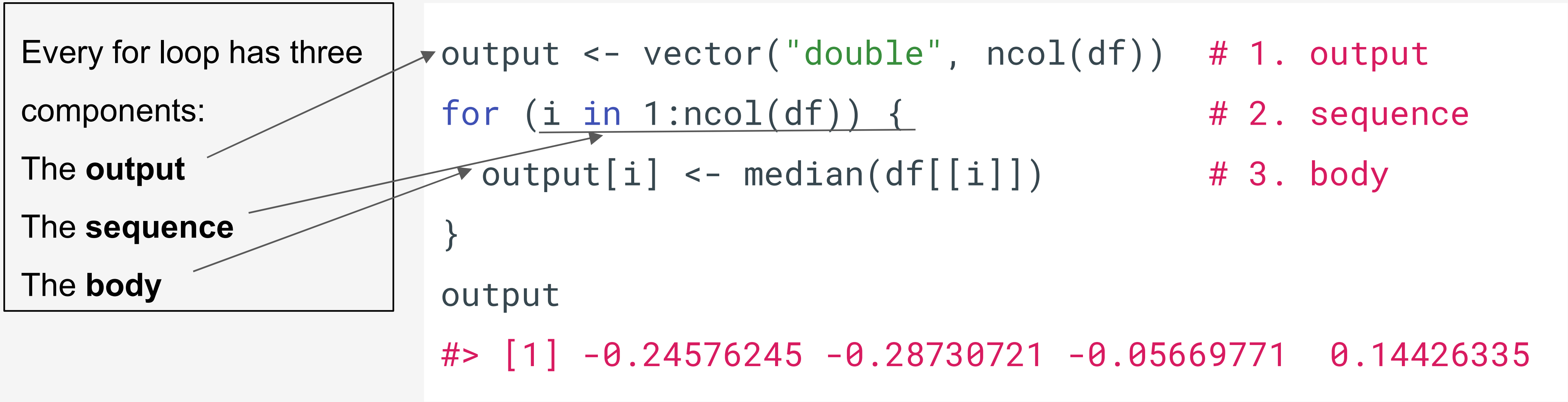
For loop

Let’s look at a simpler example...



```
for(i in 1:5){
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

for loop



Every for loop has three components:

The **output**

The **sequence**

The **body**

```r
output <- vector("double", ncol(df))  # 1. output
for (i in 1:ncol(df)) {               # 2. sequence
  output[i] <- median(df[[i]])        # 3. body
}
output
#> [1] -0.24576245 -0.28730721 -0.05669771  0.14426335
```

for loop

Before we start the loop we need to allocate sufficient space for the output (if not it can be very slow).

A general way of creating an empty vector of given length is the vector() function. It has two arguments: the type of the vector (“logical”, “integer”, “double”, “character”, etc) and the length of the vector.

```
output <- vector("double", ncol(df))  # 1. output
```

The vector data type

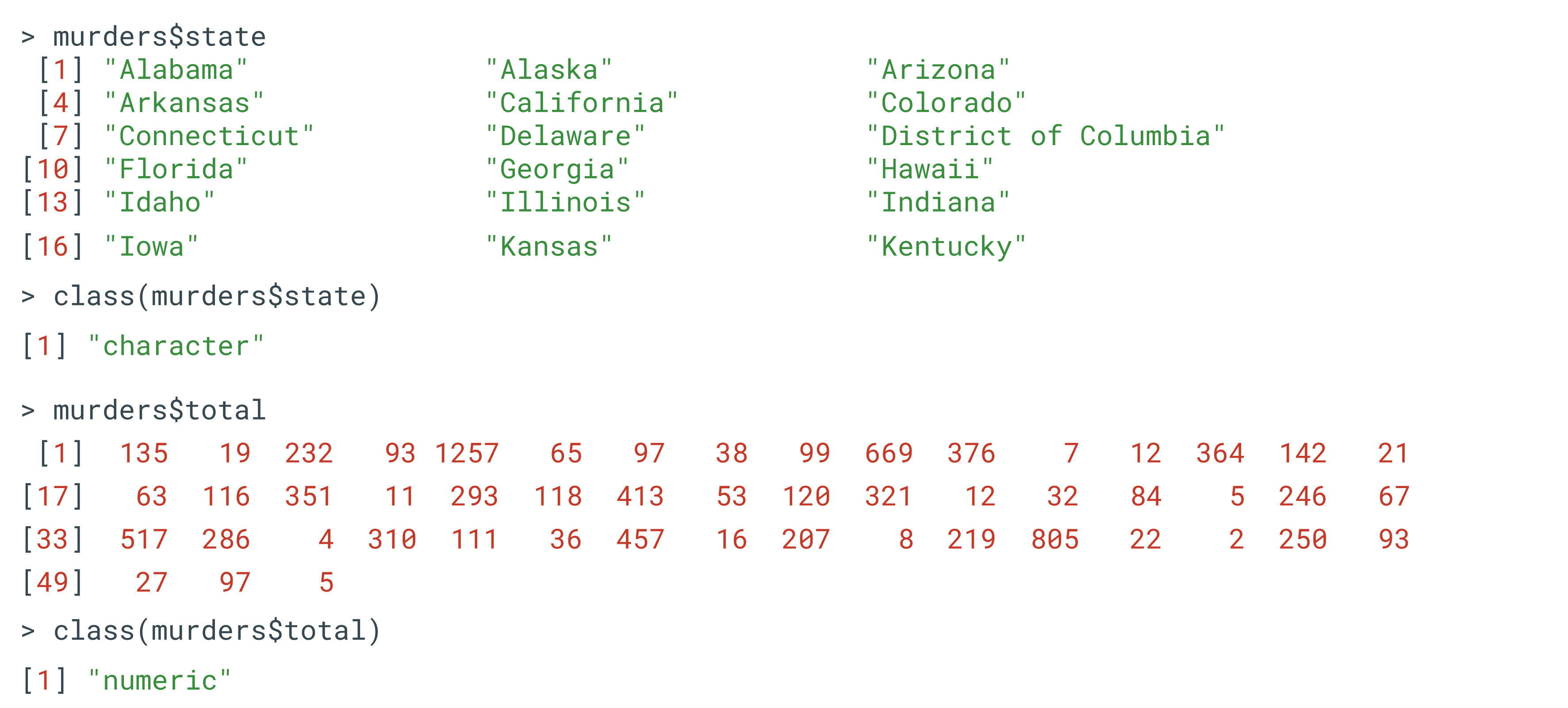
In R a vector is a kind of list of elements (list is actually something else in R, but never mind…). Vectors are created with the function c().

```
> x <- c(1, 2, 3)
> x
[1] 1 2 3
> x[2]
[1] 2
```

The vector data type

In R a vector is a kind of list of elements (list is actually something else in R, but never mind…). Vectors are created with the function c().

Vectors can be numeric, character, logic, and more.



```
> murders$state
 [1] "Alabama"              "Alaska"               "Arizona"
 [4] "Arkansas"             "California"           "Colorado"
 [7] "Connecticut"          "Delaware"             "District of Columbia"
[10] "Florida"              "Georgia"              "Hawaii"
[13] "Idaho"                "Illinois"             "Indiana"
[16] "Iowa"                 "Kansas"               "Kentucky"
> class(murders$state)
[1] "character"

> murders$total
 [1]  135   19  232   93 1257   65   97   38   99  669  376    7   12  364  142   21
[17]   63  116  351   11  293  118  413   53  120  321   12   32   84    5  246   67
[33]  517  286    4  310  111   36  457   16  207    8  219  805   22    2  250   93
[49]   27   97    5
> class(murders$total)
[1] "numeric"
```

Subsetting

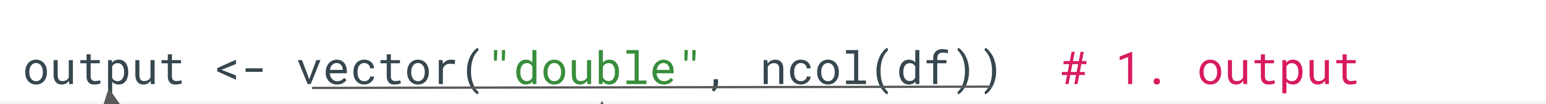
Notice the numbers in brackets in the output. These give hint about how to retrieve certain elements of a vector. This is called subsetting, or indexing (this works on many other data types in R as well).

```
> murders$state[1]
[1] "Alabama"
> murders$state[3:6]
[1] "Arizona"    "Arkansas"   "California" "Colorado"
> murders$state[c(6, 3, 5, 4)]
[1] "Colorado"   "Arizona"    "California" "Arkansas"
> murders$state[c(-6, -3, -5, -4)]
 [1] "Alabama"              "Alaska"               "Connecticut"
 [4] "Delaware"             "District of Columbia" "Florida"
...
```

for loop

Before we start the loop we need to allocate sufficient space for the output (if not it can be very slow).

A general way of creating an empty vector of given length is the vector() function. It has two arguments: the type of the vector (“logical”, “integer”, “double”, “character”, etc) and the length of the vector.



```
output <- vector("double", ncol(df))  # 1. output
```

The **result of the loop** (output) is **a vector** of type “double” (decimal numbers) and the same length as the number of columns in the data frame.

The sequence

The sequence determines **what to loop over**. (1:ncol() will generate a sequence of numbers from 1 to the number of columns in the dataframe – 1, 2, 3, 4 in this case).

“i” can be whatever character or word you like.

```
for (i in 1:ncol(df))                   # 2. sequence
```

The loop will iterate for the same number of times as there are columns in the data frame (i.e. 4 columns). “i” will be updated for every iteration (i.e. first iteration i = 1, second iteration i = 2, third i = 3 and fourth i = 4.

The body

The body is the code that does the work. It’s run repeatedly, each time with a different value for “i”.

```
output[i] <- median(df[[i]])        # 3. body
```

`df[[i]]` extracts column “i” as a vector of numbers. The function median() calculates the median of these numbers. The median is then entered into position “i” in the “output” vector. (the double brackets are needed to extract only the values in the column, and not the entire column with header).

The first iteration of the loop will be:

```
output[1] <- median(df[[1]])
```

Do Exercise 5

if statements

if statements (conditional expressions)

Conditional expressions are one of the basic features of programming. They are used for what is called *flow control*. The most common conditional expression is the if statement (or the if-else statement)

Syntax of the if-statement

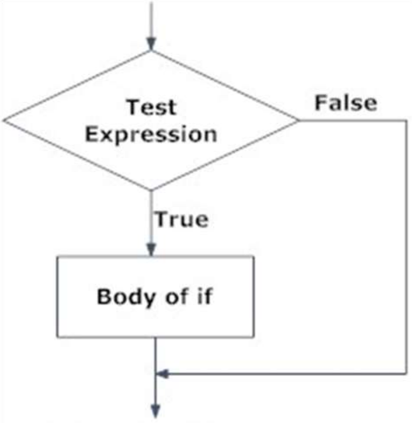


Fig: Operation of if statement

```
x <- 5
if(x > 0){
print("Positive number")
}
```

Syntax of the if-else statement

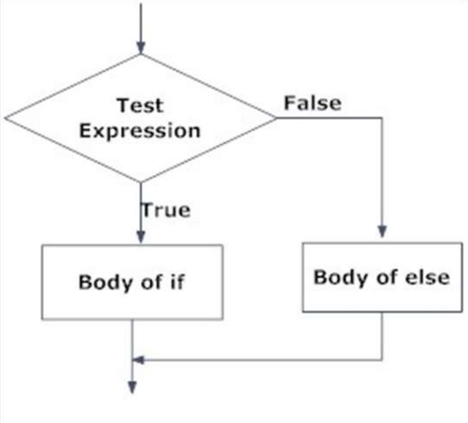
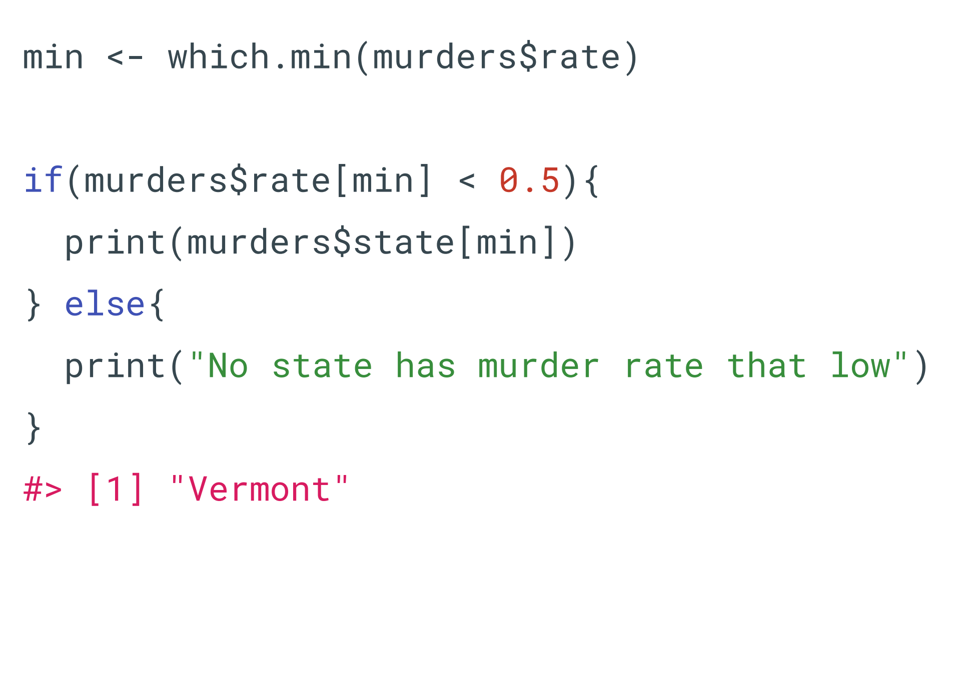


Fig: Operation of if...else statement

```
if (test_expression)
{
statement1
} else {
statement2
}
```

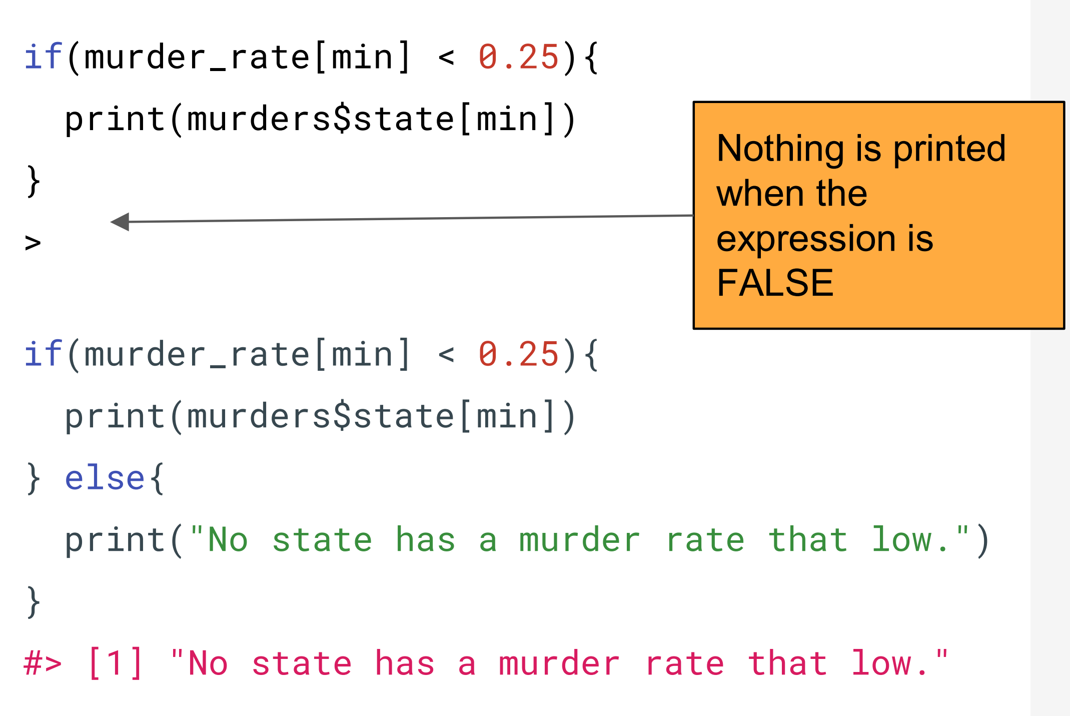
if-else statement

Here is a very simple example that tells us which states, if any, have a murder rate lower than 0.5 per 100,000. The else statement protects us from the case in which no state satisfies the condition.



```
min <- which.min(murders$rate)

if(murders$rate[min] < 0.5){
  print(murders$state[min])
} else{
  print("No state has murder rate that low")
}
#> [1] "Vermont"
```



```
if(murder_rate[min] < 0.25){
  print(murders$state[min])
}
>

if(murder_rate[min] < 0.25){
  print(murders$state[min])
} else{
  print("No state has a murder rate that low.")
}
#> [1] "No state has a murder rate that low."
```

Nothing is printed when the expression is FALSE

Do Exercise 6

See you 12:45

Running R from the command line

R scripts

The first thing we’ll do is to log on to Saga and enter your home directory.

From there type:

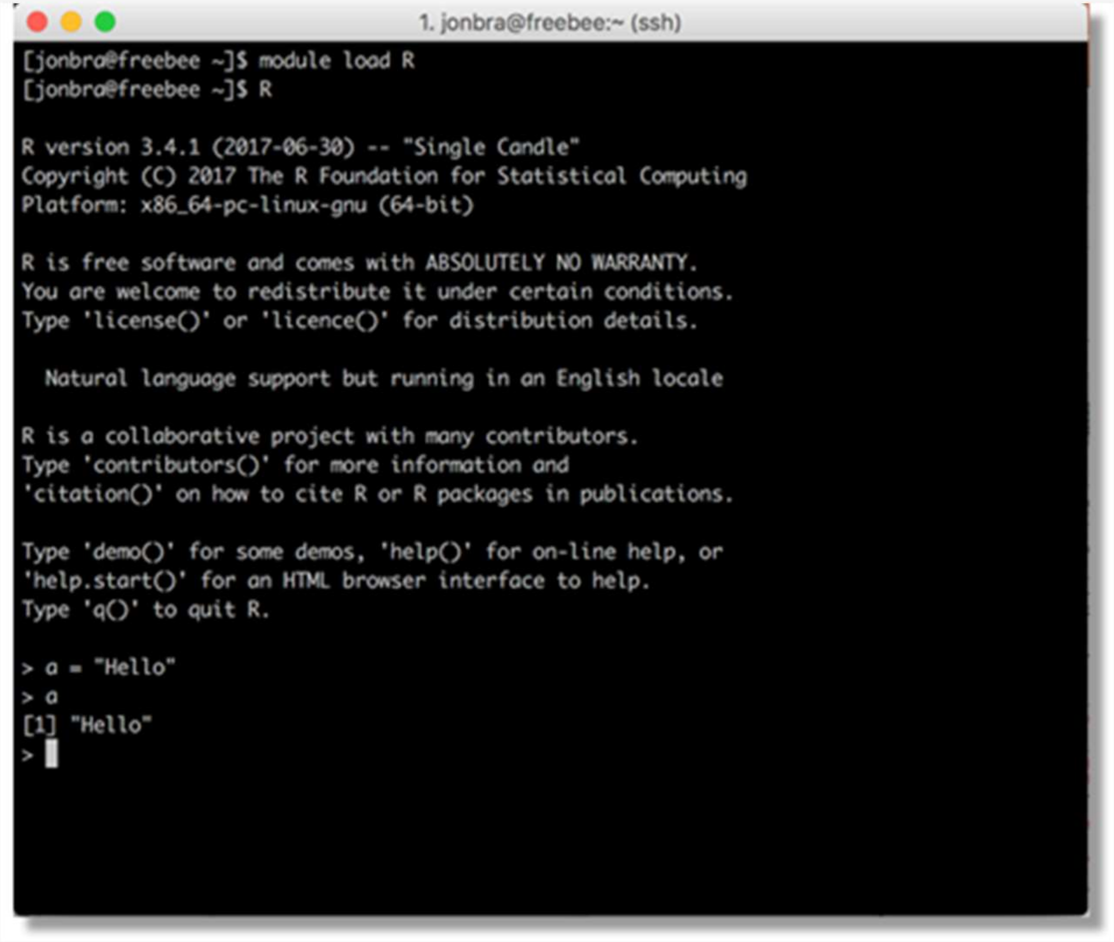
module load R/4.1.0-foss-2021a

Then start R by typing “*R*” and “Enter”.

You should see something similar to the image.

Activate tidyverse by typing

library(tidyverse)



```
[jonbra@freebee ~]$ module load R
[jonbra@freebee ~]$ R

R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> a = "Hello"
> a
[1] "Hello"
> 
```

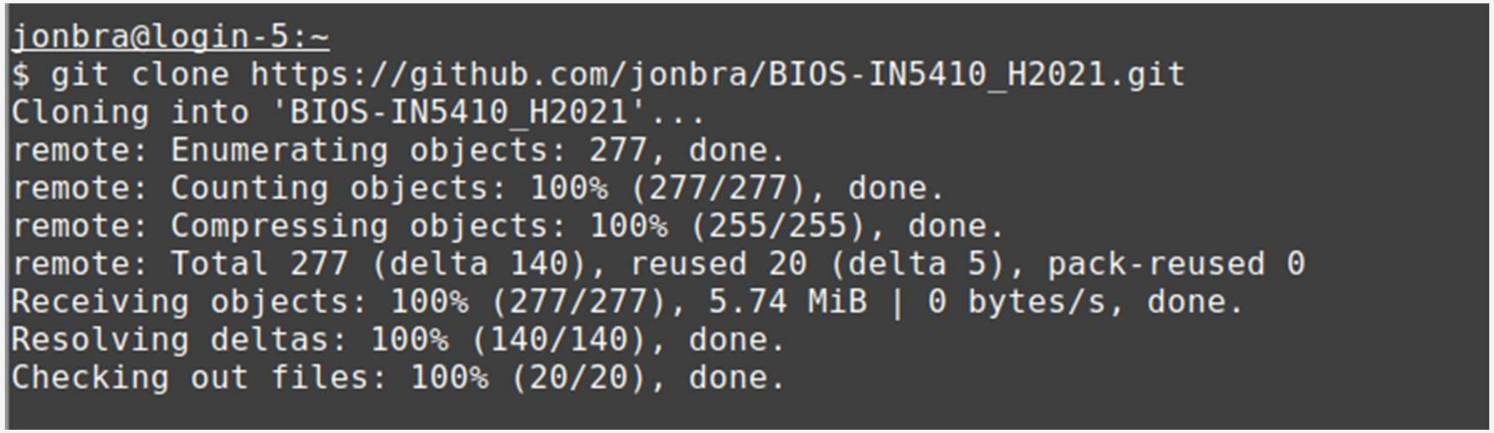
R scripts

If tidyverse is still installing, open a new Terminal window and log on to Saga again. Otherwise use the same window as before.

Then clone the BIOS-IN5410 GitHub repo (either my repo or your own copy if you created one yesterday) to your home directory by first typing *cd* and Enter, and then:

git clone https://github.com/jonbra/BIOS-IN5410_H2021.git

(NB: use the https link).



```
jonbra@login-5:~
$ git clone https://github.com/jonbra/BIOS-IN5410_H2021.git
Cloning into 'BIOS-IN5410_H2021'...
remote: Enumerating objects: 277, done.
remote: Counting objects: 100% (277/277), done.
remote: Compressing objects: 100% (255/255), done.
remote: Total 277 (delta 140), reused 20 (delta 5), pack-reused 0
Receiving objects: 100% (277/277), 5.74 MiB | 0 bytes/s, done.
Resolving deltas: 100% (140/140), done.
Checking out files: 100% (20/20), done.
```

Exercise 7

Log on to Saga and do Exercise 7.

You can try it yourself, but I will go through each part separately and explain what is going on.

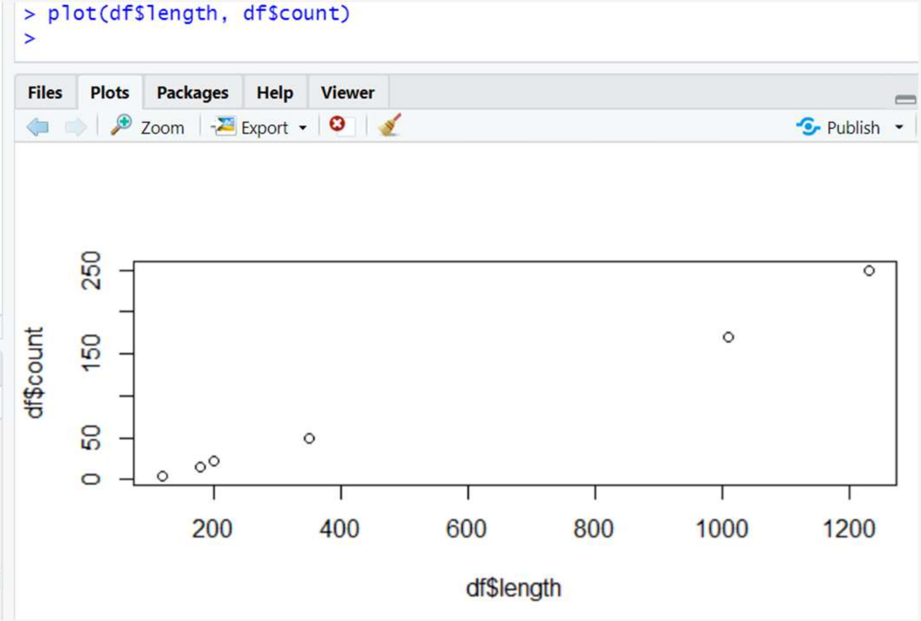
Plotting with ggplot2

Plotting with ggplot2

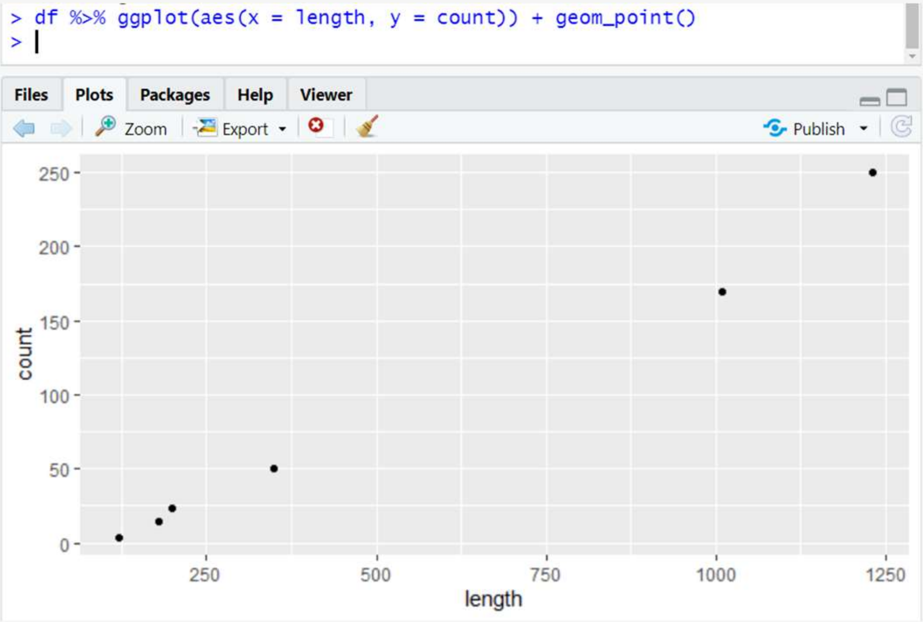
On Friday you made some simple plots with base R plotting functions.

You can make great plots with base R, don’t worry. But there’s a very popular package for plotting in R, ggplot2, that is very useful to know about.

ggplot2 is automatically activated when you load Tidyverse, and it’s particularly suited to operate on tidy data.



```
> plot(df$length, df$count)
>
```

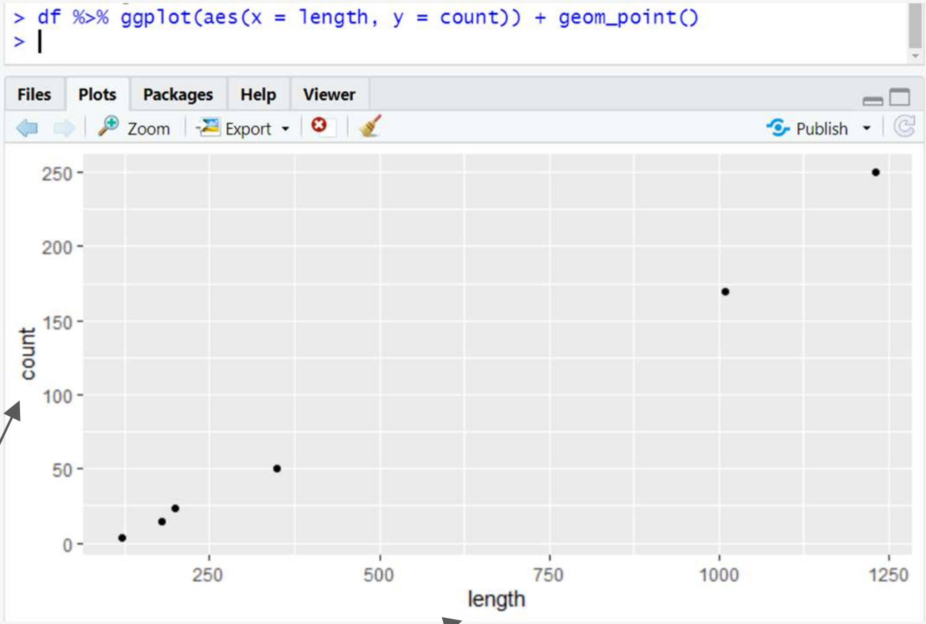


```
> df %>% ggplot(aes(x = length, y = count)) + geom_point()
> |
```

Plotting with ggplot2

```
> df
# A tibble: 6 x 3
  Gene  count length
  <chr> <dbl>  <dbl>
1 A         4    120
2 B        50    350
3 C        23    200
4 D       250   1230
5 E        15    180
6 F       170   1010

> df %>%
    ggplot(aes(x = length, y = count)) +
    geom_point()
```



```
> df %>% ggplot(aes(x = length, y = count)) + geom_point()
> |
```

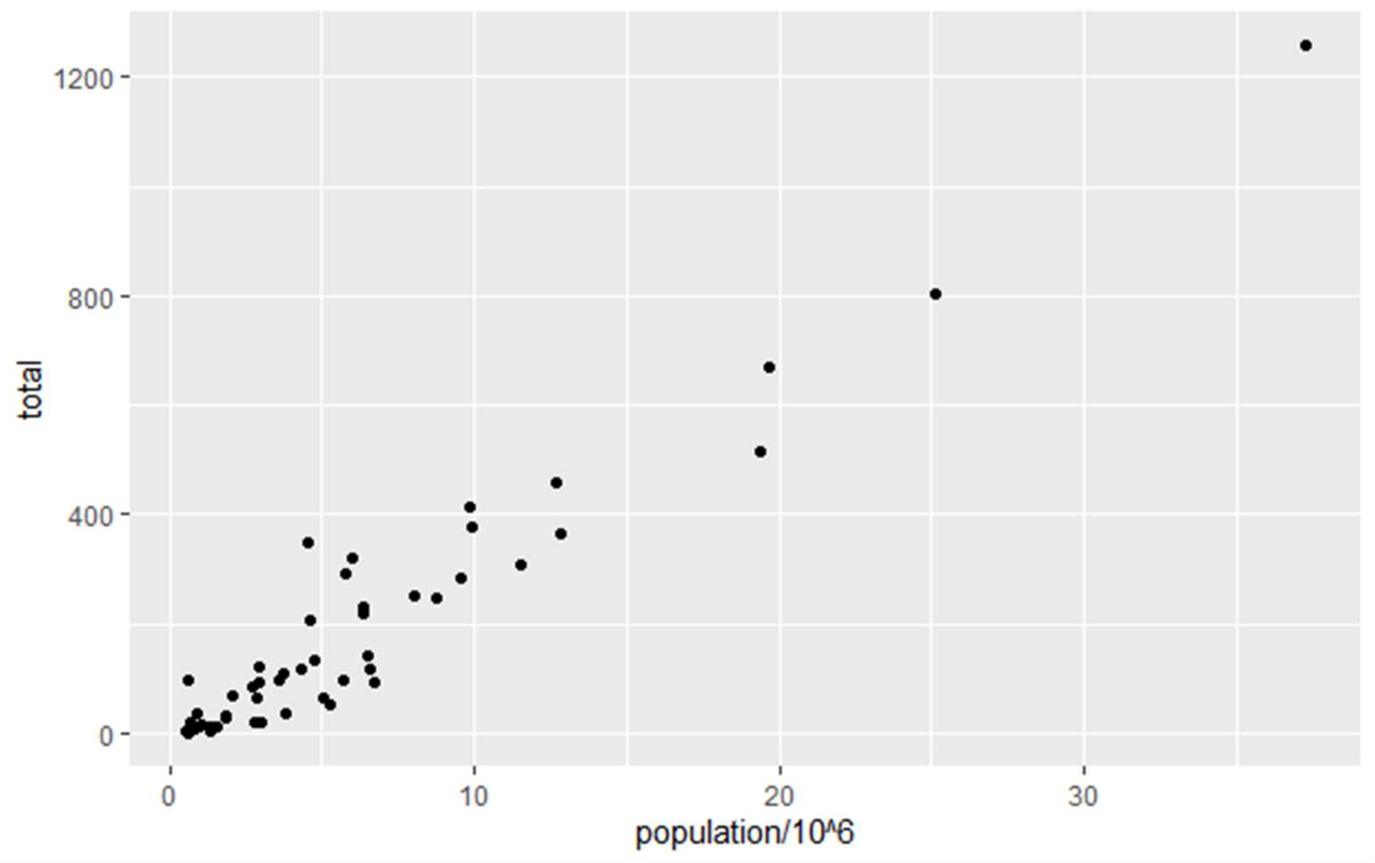
Plots are initiated with the function **ggplot()**. Then the different subfunctions are tied together in layers using the “+” symbol (like a “pipe”).

Aesthetics (aes) is a mapping of the variables in the data the different properties of the plot (the geom), like x and y axes, color, etc.

Plotting with ggplot2

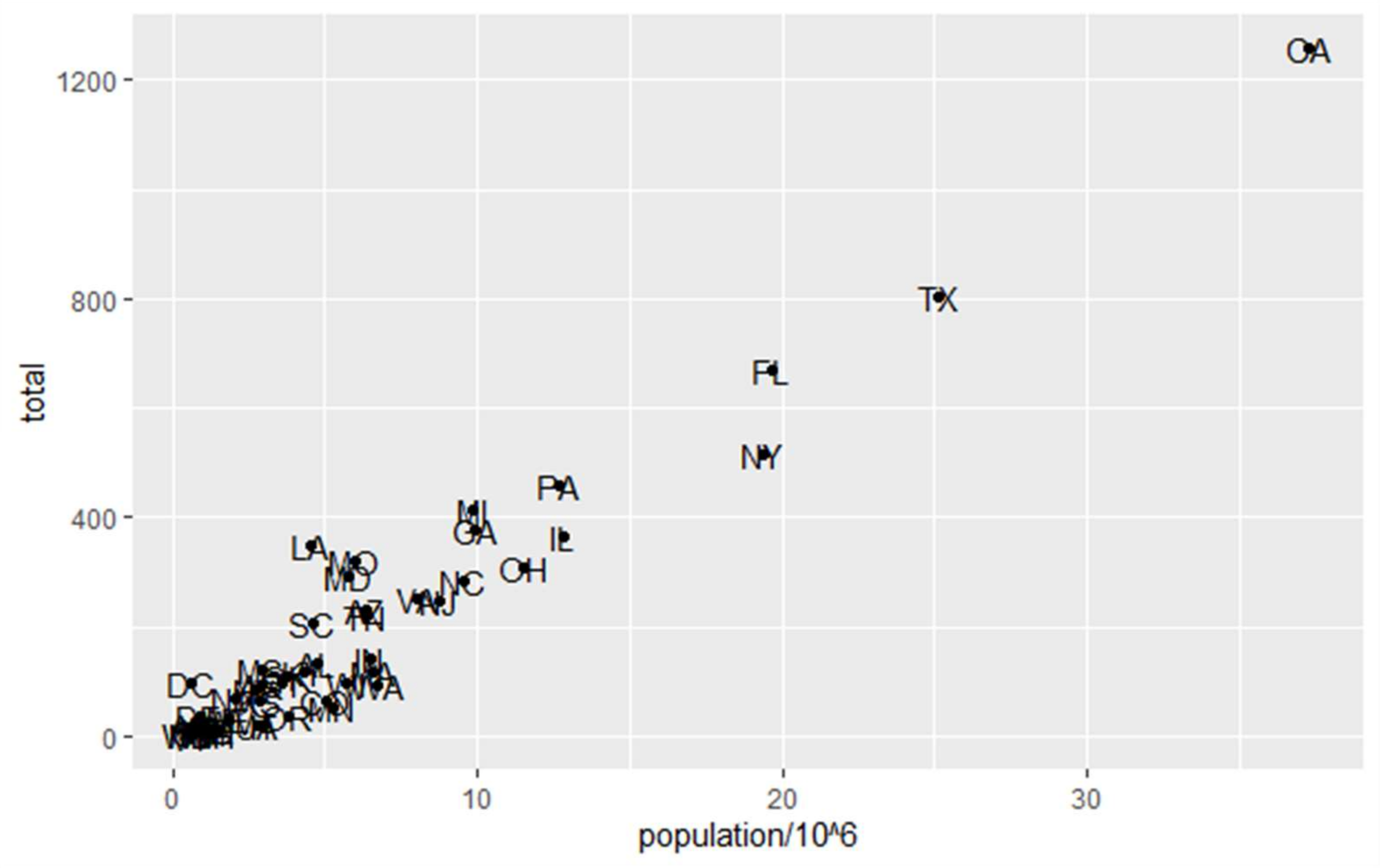
A quick demonstration of how ggplot2 plots are built up by adding layers

```
murders %>% ggplot() +
  geom_point(aes(x =
population/10^6, y = total))
```



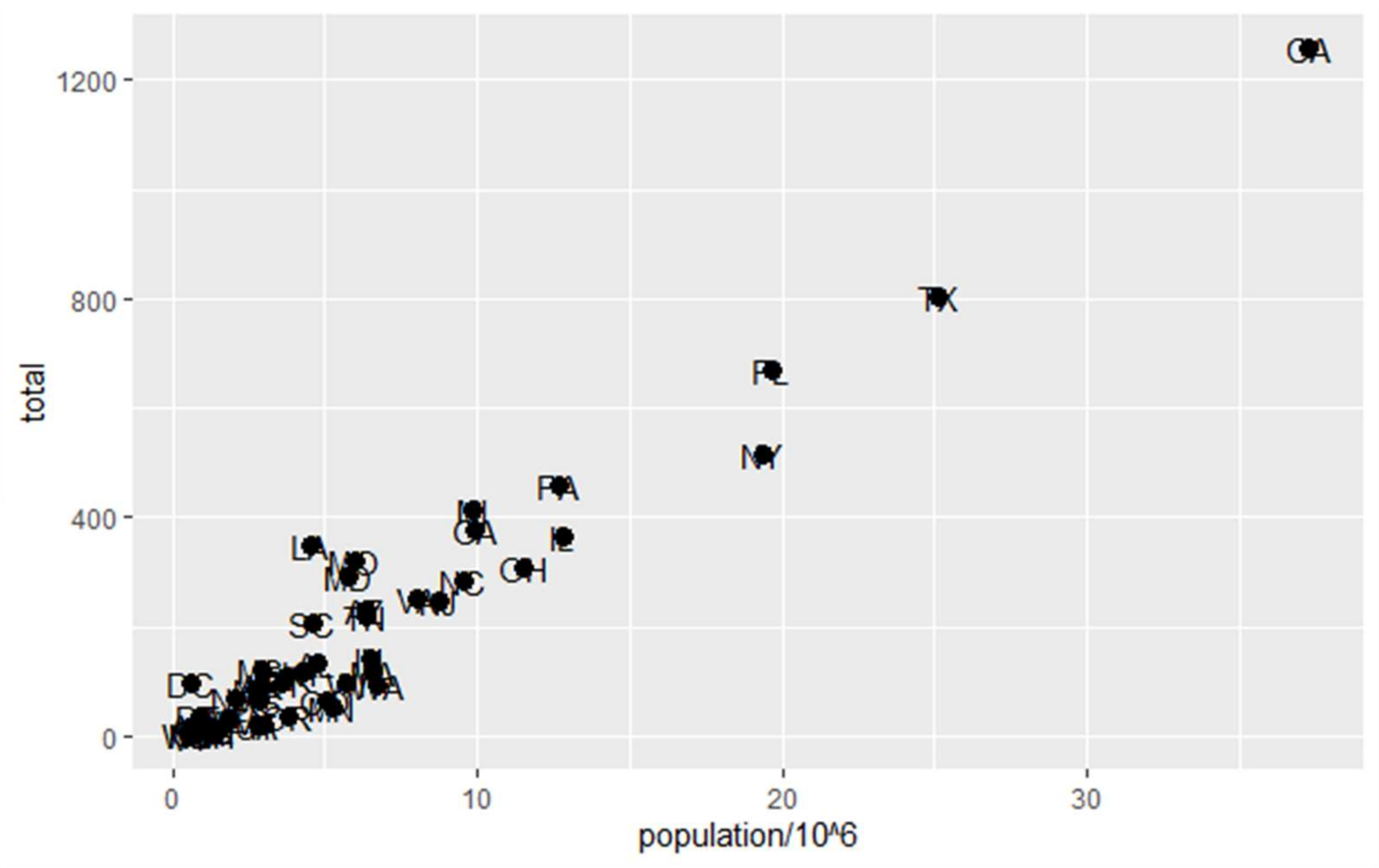
Plotting with ggplot2

```
murders %>% ggplot() +
  geom_point(aes(x =
population/10^6, y = total)) +
  geom_text(aes(population/10^6,
total, label = abb))
```



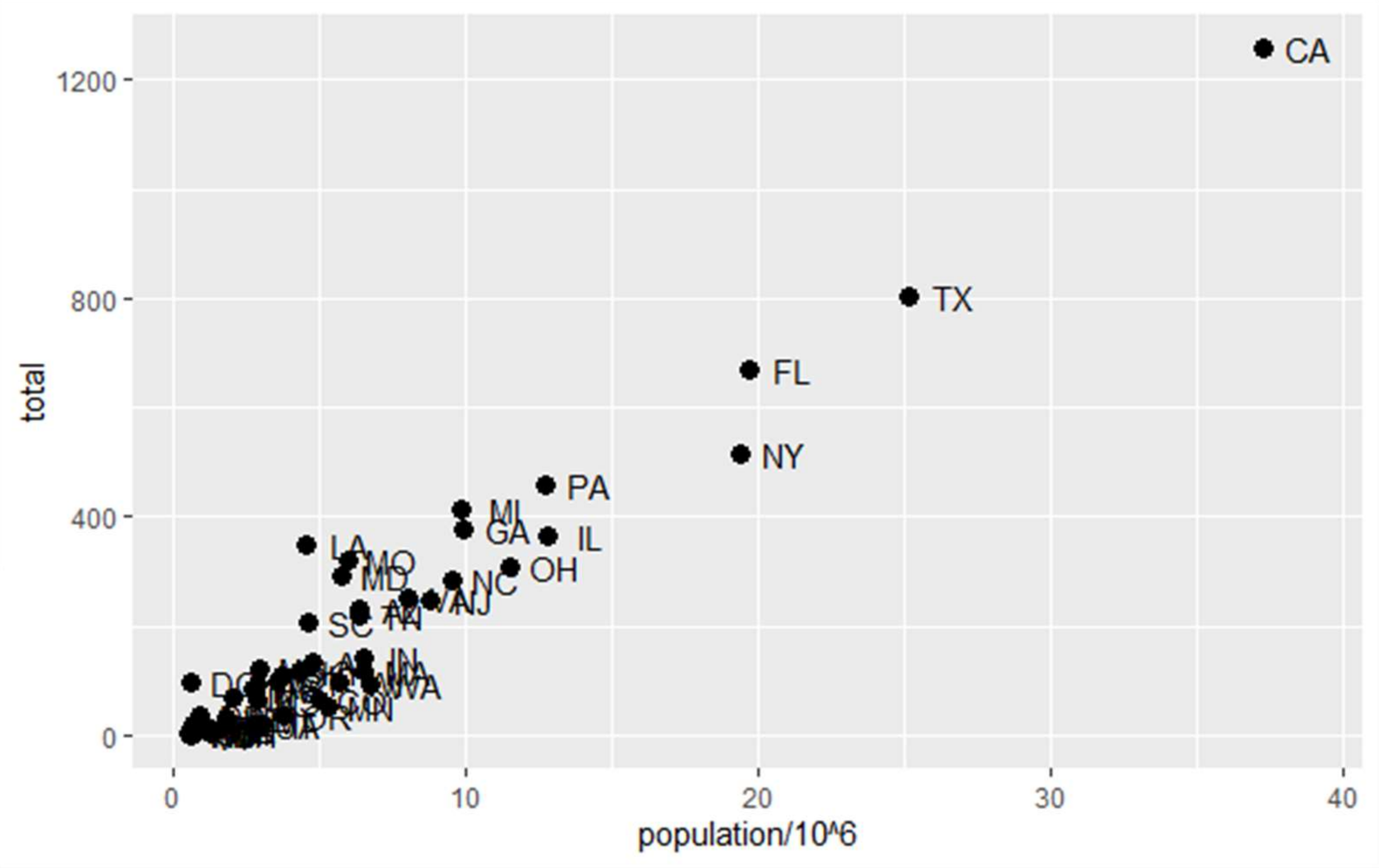
Plotting with ggplot2

```
murders %>% ggplot() +
  geom_point(aes(x =
population/10^6, y = total), size
= 3) +
  geom_text(aes(population/10^6,
total, label = abb))
```



Plotting with ggplot2

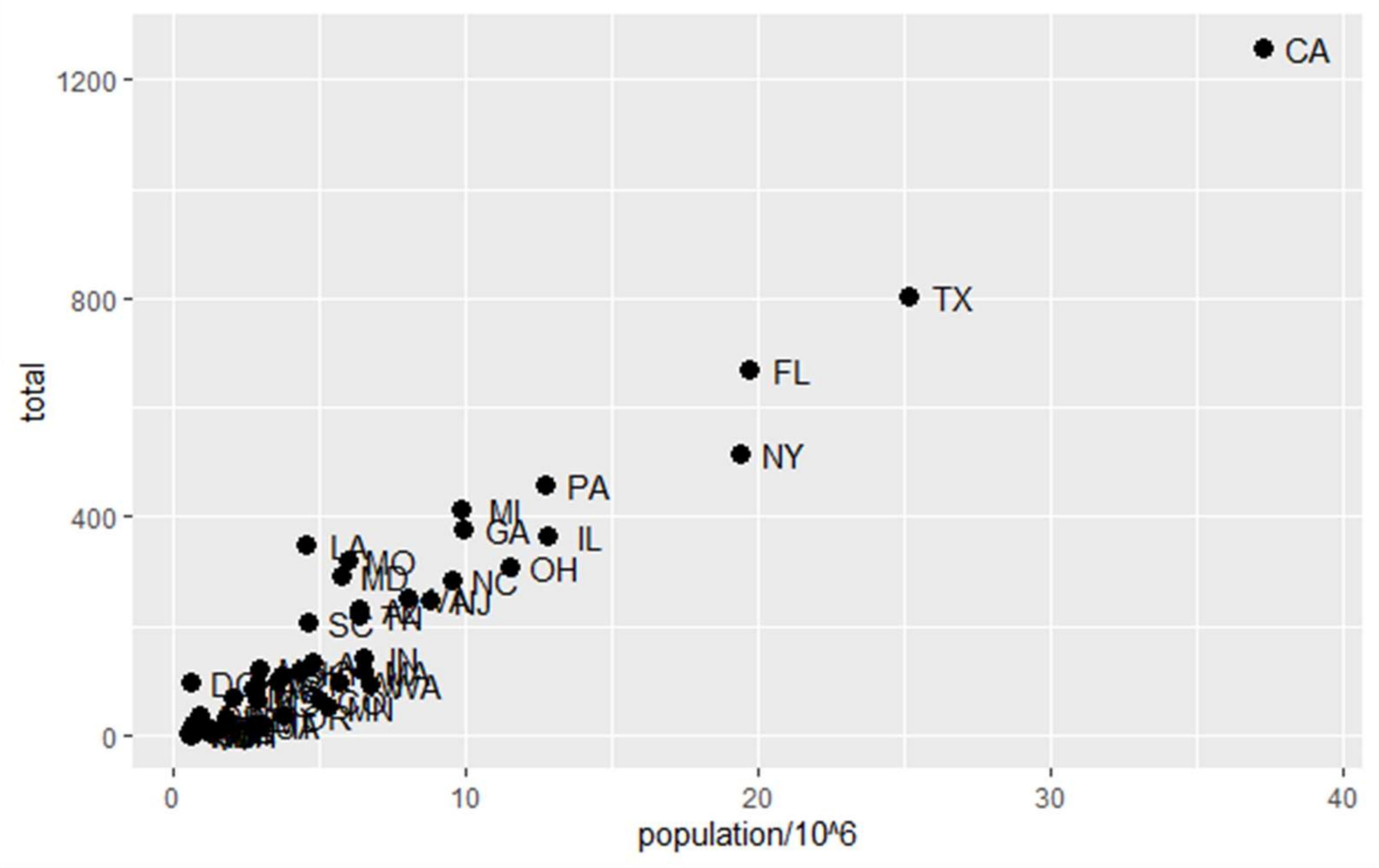
```
murders %>% ggplot() +
  geom_point(aes(x =
population/10^6, y = total), size
= 3) +
     geom_text(aes(population/10^
6, total, label = abb), nudge_x =
1.5)
```



Plotting with ggplot2

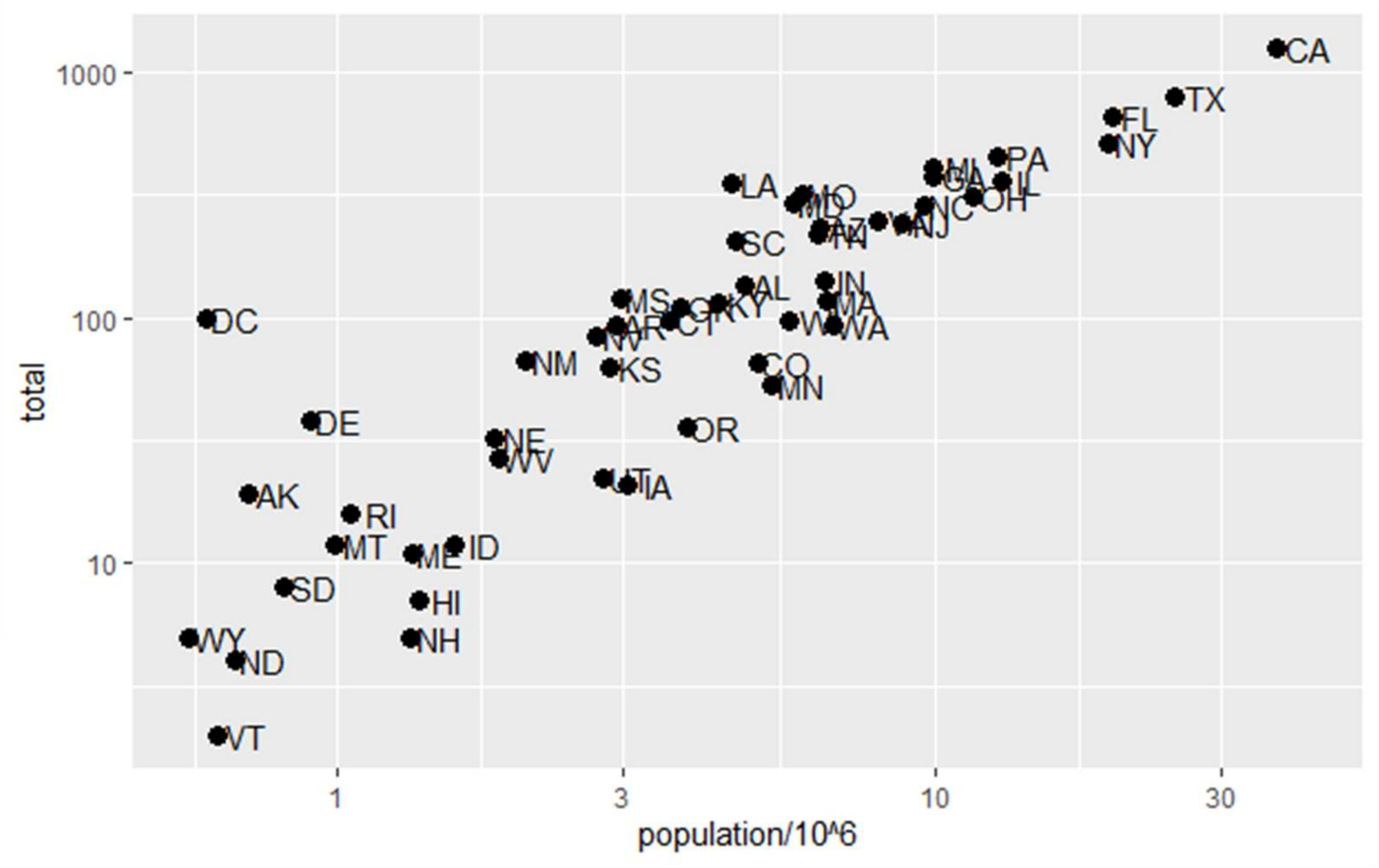
Global aesthetics. Apply to all layers

```
murders %>%
  ggplot(aes(population/10^6,
total, label = abb)) +
  geom_point(size = 3) +
  geom_text(nudge_x = 1.5)
```



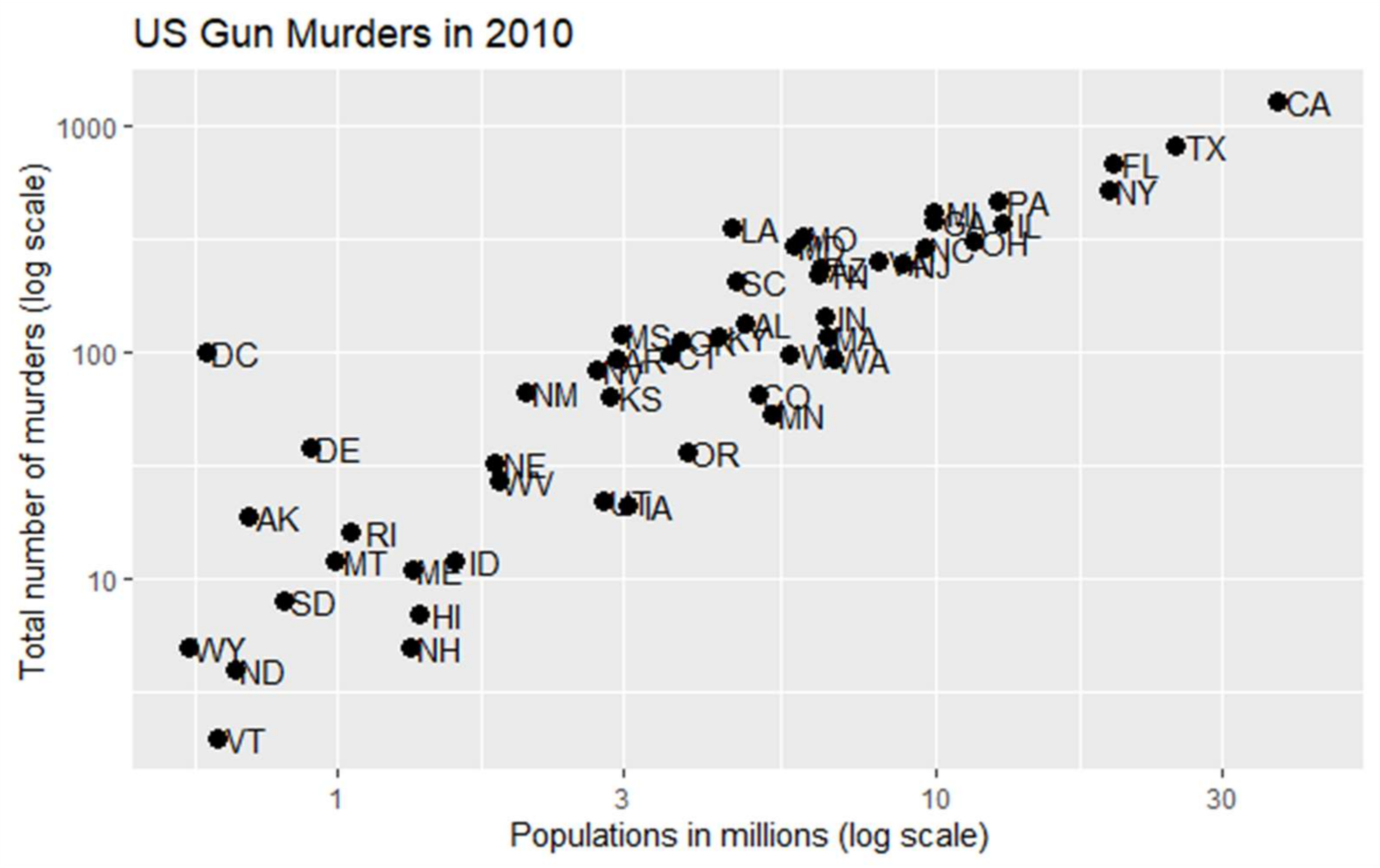
Plotting with ggplot2

```
murders %>%
  ggplot(aes(population/10^6,
total, label = abb)) +
  geom_point(size = 3) +
  geom_text(nudge_x = 0.05) +
  scale_x_continuous(trans =
"log10") +
  scale_y_continuous(trans =
"log10")
```



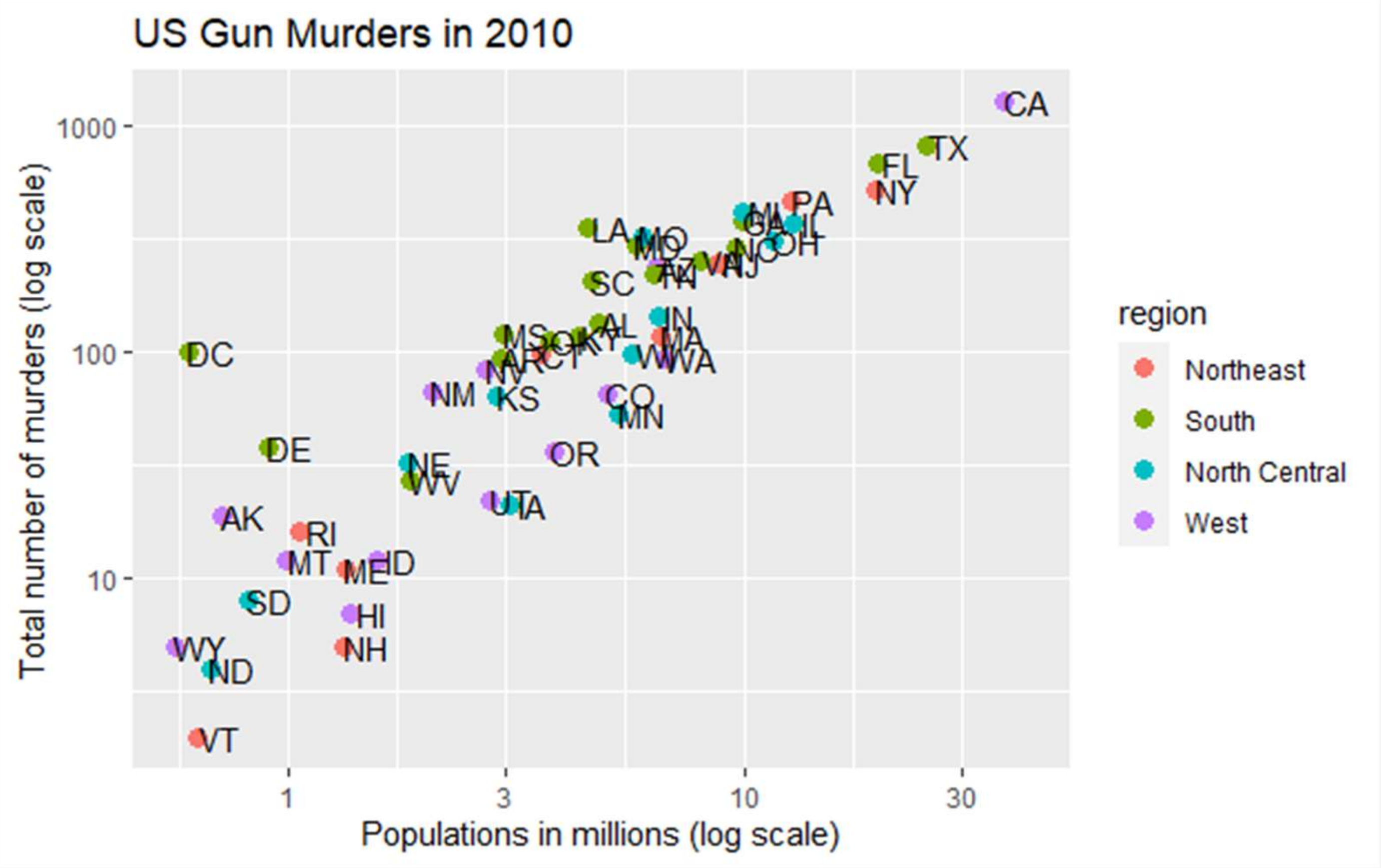
Plotting with ggplot2

```
murders %>%
  ggplot(aes(population/10^6,
total, label = abb)) +
  geom_point(size = 3) +
  geom_text(nudge_x = 0.05) +
  scale_x_continuous(trans =
"log10") +
  scale_y_continuous(trans =
"log10") +
  xlab("Populations in millions
(log scale)") +
  ylab("Total number of murders
(log scale)") +
  ggtitle("US Gun Murders in
2010")
```



Plotting with ggplot2

```
murders %>%
  ggplot(aes(population/10^6, total,
label = abb)) +
  geom_point(aes(col = region), size
= 3) +
  geom_text(nudge_x = 0.05) +
  scale_x_continuous(trans =
"log10") +
  scale_y_continuous(trans =
"log10") +
  xlab("Populations in millions (log
scale)") +
  ylab("Total number of murders (log
scale)") +
  ggtitle("US Gun Murders in 2010")
```



Do Exercise 8