Fundamentals of R Programming

Dr. Liu Qizhang

07 Jan 2019

Introduction

This file gives you an introduction of the fundamentals needed for you to write your first codes using R.

Knowing the Working Environment

In programming, the working environment refers to the settings for your codes to work. You must know where to store, compile and run your code. Below are some definitions that you need to get familiar with:

Workspace: the workspace refers to all the variables and functions (collectively called *objects*) that you create during an R session, as well as any packages that are loaded. When you terminate an R session, you have an option to save an image of the current workspace, which will be automatically reloaded the next time R is started.

Working Directory: it is where R will look, by default, for files you ask it to load and will, by default, store your files to.

Project: Data, R scripts, analytical results, and figures about a particular problem are normally organised and stored under one project. Use *File -> New Project* menu to create a new project or *File–>Open Project* menu to open an existing project.

Useful functions

You can check your working directory with:

```
getwd()
```

```
## [1] "C:/Users/bizlqz/Dropbox/Qizhang materials/DBA3702 Descriptive Analytics with R/Lectur
e Notes/2 - R Basics"
```

You can set your working directory to other diretory as shown in the following examples. Note that ".." will go one level up the current directory.

```
#Set the working directory to a sub folder of current working directory. The sub folder must
 already exist
setwd("Test")
getwd()
```

```
## [1] "C:/Users/bizlqz/Dropbox/Qizhang materials/DBA3702 Descriptive Analytics with R/Lectur
e Notes/2 - R Basics/Test"
```

```
#Set the working directory to the parent folder of the current working diretory
setwd("..")
getwd()
```

```
## [1] "C:/Users/bizlqz/Dropbox/Qizhang materials/DBA3702 Descriptive Analytics with R/Lectur
e Notes"
```

In RStudio, you can use *Tools–>Global Options* to change the default working directory.

You can list all the objects in the current work space by ls() command:

```
ls()
```

```
## character(0)
```

Object here means variables and functions that you have defined when working in the current work space. For example, the following command defines a variable named *testName*. After running this command, try ls() again and you can find *testName* in the list.

```
testName = "ABC"
ls()
```

```
## [1] "testName"
```

You can use *rm(x)* command to remove object "x" from the work space, or use *rm(list=ls())* to remove all objects from the work space. As it is not recoverable, be cautious when using these commands.

dir() command will allow you to list all the files and sub folders in the current working directory.

```
dir()
```

```
##  [1] "61428706.tmp"
##  [2] "Class Discussion - Topic 1.Rmd"
##  [3] "Class_Discussion_-_Topic_1.docx"
##  [4] "Class_Discussion_-_Topic_1.html"
##  [5] "Data"
##  [6] "Files"
##  [7] "Files.zip"
##  [8] "Fundamentals of R Programming.Rmd"
##  [9] "Fundamentals_of_R_Programming.html"
## [10] "Fundamentals_of_R_Programming.Rmd"
## [11] "HR Analysis.R"
## [12] "HR_data.csv"
## [13] "HR_data.xlsx"
## [14] "HR_Data_New.csv"
## [15] "HR_Data_New.xlsx"
## [16] "Lesson 1 slides.pptx"
## [17] "Prof Liu Video Script - Lesson 1 (14 Sept).doc"
## [18] "Prof Liu Video Script - Lesson 1.doc"
## [19] "Test"
## [20] "Topic 1 Learning Materials.docx"
## [21] "Topic 1 Learning Materials.pdf"
## [22] "Video Scripts.docx"
```

One of the best features of R function is its scalability and flexibility coming with a wide range of parameters. Mastering R is a process of playing around with various combinations of parameters in R functions. You may use dir() as an example to explore the power of this function with support of its parameters. To find detailed description of a particular function, just type “?” followed by the function name in the console. For example: *?dir* will give you a help file on using dir function. In case the function you ask for does not exist in your work space (maybe because the corresponding package is not loaded), then you can use “??” followed by the function name to further check.

You can also use another function list.files() to list all the files and folders in a directory. One thing that makes R the top choice of many data scientists and programmers is the availability of options to solve a problem. However, it could be painful for beginners as well because they can easily get confused. My advice is that you just choose the one that suits your needs and stick to it. If you have time, search online or run the options yourself to compare them, then choose the best.

```
list.files()
```

The following functions are also useful if you need to work with files and folders in R.

```
#Check if a file exists
file.exists("test.R")
```

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

```
#Check if the folder "Data" exists. If not, create it
ifelse(!dir.exists("Data"),dir.create("Data"),"Folder exists already!")
```

```
## [1] "Folder exists already!"
```

Atomic Data Types

Learning coding for any programming language normally starts with knowing the data types defined in the language. R is not an exception. R has five basic (or *atomic*) classes of objects:

Logical

A logical value is normally produced by some logical operations. For example,

```
#Compare if 2 is more than 3, and then pass the result to variable x
x <- 2>3
x
```

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

You can use logical operators such as “&” (and), “|” (or), and “!” (negation) to do more complex logical operations. For example,

```
(3>2) & (7>5)
```

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

```
(3>2) | (5>7)
```

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

```
!(3>2)
```

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

Numeric

Decimal values are called numerics in R. It is the **default** computational data type. If we assign a decimal value to a variable x as follows, x will be of numeric type.

```
x<-1.23
class(x)
```

```
## [1] "numeric"
```

Furthermore, even if we assign an integer to a variable k, it is still being saved as a numeric value.

```
k<-2
class(k)
```

```
## [1] "numeric"
```

Integer

In order to create an integer variable in R, we need to use the as.integer function.

```
x<-as.integer(2)
class(x)
```

```
## [1] "integer"
```

You can also cast a numeric value to be integer, however, some value wil be lost as R will truncate it to an integer.

```
x<-as.integer(2.34)
x
```

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

```
#notice that it is not rounded up to 3 but instead truncated to 2
x<-as.integer(2.56)
x
```

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

You can also cast a logical value to be integer, where “TRUE” will become 1 and “FALSE” will become 0.

```
x<-as.integer(5>6)
x
```

```
## [1] 0
```

However, you cannot cast a text string to either integer or numeric.

```
as.numeric("abc")
```

```
## Warning: NAs introduced by coercion
```

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

Character

A character object is used to represent string values in R. For example, “James”, “China”, etc. The following are some common functions used on character objects:

- *nchar()* is used to get the length of a text string (i.e. the number of characters in the string).

```
a<-"Singapore"
nchar(a)
```

```
## [1] 9
```

- *regexpr* is used to find the starting position of a small string in a large string. It will return -1 if no matching is found.

```
regexpr("ex", "longtext")
```

```
## [1] 6
## attr(,"match.length")
## [1] 2
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

```
regexpr("exs","longtext")
```

```
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

- *gregexpr* is used to find positions of every match of a small sting in a large string.

```
gregexpr("a","banana")
```

```
## [[1]]
## [1] 2 4 6
## attr(,"match.length")
## [1] 1 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

```
#notice the difference in return values
regexpr("a","banana")
```

```
## [1] 2
## attr(,"match.length")
## [1] 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

- *grep* is used to find the positions of a regular expression in a vector of text strings. Here vector refers to a list of objects. We are going to discuss about vector later.

```
txt<-c("arm","foot","lefroo", "bafoobar")
grep("foo", txt)
```

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

- *substr* is used to extract part of a text string based on position in the text string.

```
#Get the sub string of "Singapore" from the 2nd character to the 4th character
substr("Singapore",2,4)
```

```
## [1] "ing"
```

- *sub* is used to replace the first match of a string with a new string.

```
#Replace "or" in Singapore with "es"
sub("or","es","Singapore")
```

```
## [1] "Singapese"
```

- *gsub* will replace every match of a given sub string in a string with a new string.

```
#Replace every "a" in "banana" with "o"
gsub("a","o","banana")
```

```
## [1] "bonono"
```

Complex

A complex value in R is defined via the pure imaginary value *i*. For example,

```
x<-1-2i
class(x)
```

```
## [1] "complex"
```

Note that complex operation works only with complex value. For example, the following statement will not work.

```
sqrt(-1)    #try to find the square root of -1
```

```
## Warning in sqrt(-1): NaNs produced
```

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

Instead, you need to first convert -1 to a complex value first.

```
sqrt(-1+0i)   #alternatively, you can use sqrt(as.complex(-1))
```

```
## [1] 0+1i
```

Data Structures

Just like the three primary colours Red, Green, and Blue form our colourful world, with the above five atomoic data types, we will be able to work with all sorts of data, thanks to the data structures provided in R. A **data structure** is a specialised format for organising and storing data. R provides 5 data structures: Vector, List, Matrix, Data Frame, and Array, which can be classified by their *dimentionality* (1d, 2d, or nd) and whether they are *homogeneous* (all elements are of the same data type) or *heterogeneous* (they may contain elements of different data types), as shown in table below:

	Homogeneous	**Heterogeneous**
1d	Vector	List
2d	Matrix	Data Frame
nd	Array	

Almost all other objects are built on these data structures. Note that R does not have 0-dimensional object. Integer or string, for example, are treated as vectors of length one.

Vector

Vector is the most basic data structure. It stores an ordered array of elements of the same data type. The easiest way to create a vector is using *c()* function. The following example creates a vector with three given elements.

```
a<-c(3,1,5)
a
```

```
## [1] 3 1 5
```

All the elements of a vector must be of the same data type. In case you want to combine different types in one single vector, they will be **coerced** to the most flexible type. For example,

```
a<-c("Name",1)
a       #1 is coerced as character
```

```
## [1] "Name" "1"
```

You can check the type of a vector by using *typeof()* or check if the vector is of a specific type by using functions like *is.character()*, *is.logical()*, etc.

```
a<-c(1,3,4)
typeof(a)
```

```
## [1] "double"
```

Vector Naming

You can also name your vectors if necessary in 3 different ways:

```
a<-c(1,3,4)
furniture <- c("desks", "tables", "chairs")
names(a) <- furniture
a
```

```
##  desks tables chairs
##      1      3      4
```

```
a<-c("desks" = 1, "tables" = 3, "chairs" = 4)
a
```

```
##  desks tables chairs
##      1      3      4
```

```
a<-c(desks = 1, tables = 3, chairs = 4)
a
```

```
##  desks tables chairs
##      1      3      4
```

```
a<-c(TRUE, FALSE, TRUE)
is.logical(a)
```

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

Vector Coercion

You can manually coerce the type of a vector, if necessary. For example,

```
a<-c(TRUE, FALSE, TRUE)
a<-as.integer(a)
a
```

```
## [1] 1 0 1
```

```
mean(a)
```

```
## [1] 0.6666667
```

There are some built in functions in R for us to conveniently generate some basic vectors. For example,

```
#The parameters used here for seq() are the start,stop,step
seq(1,9,2)             #for examples of seq() function, use ?seq
```

```
## [1] 1 3 5 7 9
```

```
rep(c(2,3,4), 3)       #Use ?rep to see more examples of rep() function
```

```
## [1] 2 3 4 2 3 4 2 3 4
```

Vector Arithmetic

Vector Arithmetic is element-wise regardless of computation with scalars or with other vectors. Below are common vector to scalar computations.

```
earnings <- c(50, 100, 30)
earnings + 100
```

```
## [1] 150 200 130
```

```
earnings - 20
```

```
## [1] 30 80 10
```

```
earnings * 3
```

```
## [1] 150 300  90
```

```
earnings / 10
```

```
## [1]  5 10  3
```

```
earnings ^ 2
```

```
## [1]  2500 10000   900
```

Unlike traditional mathematics, vector to vector multiplication and division in R *does not* result in single scalars or matrices but instead results in a vector of the original size.

```
expenses <- c(30, 40, 80)
earnings - expenses
```

```
## [1]  20  60 -50
```

```
earnings + c(10, 20, 30)
```

```
## [1]  60 120  60
```

```
earnings * c(1, 2, 3)
```

```
## [1]  50 200  90
```

```
earnings / c(1, 2, 3)
```

```
## [1] 50 50 10
```

```
earnings > expenses
```

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

Some other Vector Element Arithmetics

```
#The summation of our earnings
sum(earnings)
```

```
## [1] 180
```

```
#Our average daily earnings
mean(earnings)
```

```
## [1] 60
```

```
#The product of our earnings
prod(earnings)
```

```
## [1] 150000
```

Vector Subsetting

Subsets of vectors can be found using "[]" square brackets using either the elements index or its name (if named). Note: R uses 1-based indexing, compared to Python's 0-based indexing.

```
materials <- c(wood = 17, cloth = 36, silver = 24, gold = 3)
materials[1]            #In python you would get cloth 36 instead
```

```
## wood
##   17
```

```
materials["wood"]
```

```
## wood
##   17
```

```
materials[3]
```

```
## silver
##     24
```

```
materials["silver"]
```

```
## silver
##     24
```

Multiple elements can be selected, wherein order matters

```
materials[c(4,3)]
```

```
##   gold silver
##      3     24
```

```
metals <- materials[c(3,4)]
metals
```

```
## silver   gold
##     24      3
```

```
materials[c("silver", "gold")]
```

```
## silver   gold
##     24      3
```

Sometimes when there are more to keep in the subset than not, using "-" is easier

```
moreMetals <- c(copper = 14, bronze = 31, silver = 24,
                gold = 3, platinum = 1, palladium = 0)
moreMetals[c(1,2,3,4,5)]
```

```
##   copper   bronze   silver     gold platinum
##       14       31       24        3        1
```

```
moreMetals[-6]
```

```
##   copper   bronze   silver     gold platinum
##       14       31       24        3        1
```

```
#moreMetals[-"palladium"] does not work and throws an error however
```

Subsetting can also be done with a logical vector the same length or less than the vector to be subset.

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

```
##   copper   silver platinum
##       14       24        1
```

```
moreMetals[c(TRUE,FALSE)] #Note the recycling taking place
```

```
##   copper   silver platinum
##       14       24        1
```

```
moreMetals[c(TRUE,FALSE,TRUE,FALSE)]  #Where these three are essentially the same
```

```
##   copper   silver platinum
##       14       24        1
```

```
moreMetals[c(TRUE,FALSE,TRUE)] #however this will be different
```

```
##    copper    silver      gold palladium
##        14        24         3         0
```

Longer logical vectors will introduce NAs

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

```
##   copper   silver platinum     <NA>
##       14       24        1       NA
```

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

```
##   copper   silver platinum     <NA>     <NA>
##       14       24        1       NA       NA
```

Matrix

Matrix is a vector of elements arranged in two dimensions. It can be formed by introducing *dim()*. Matrix is the data structure in R corresponding to matrix in linear algebra. As many theories in advanced statistics are developed on linear algebra, matrix plays an important role in R as well. The following examples show various ways of creating matrices in R:

```
m1<-matrix(3:8,ncol=3,nrow=2)
m1
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    7
## [2,]    4    6    8
```

```
m1<-matrix(3:8,nrow=2)  #R can infer the other dimension
m1
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    7
## [2,]    4    6    8
```

```
m1<-matrix(3:8,ncol=3)  #As long as 1 dimension is specified
m1
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    7
## [2,]    4    6    8
```

```
m2<-3:8                #m2 is a vector
m2
```

```
## [1] 3 4 5 6 7 8
```

```
dim(m2)<-c(3,2)        #Assign a dimension to m2 to turn it into a matrix with 3 rows and 2
 columns
m2
```

```
##      [,1] [,2]
## [1,]    3    6
## [2,]    4    7
## [3,]    5    8
```

```
dim(m2)<-c(2,3)        #It can be turned into a matrix with 2 rows and 3 columns
m2
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    7
## [2,]    4    6    8
```

Matrix, like Vector also allows for recycling, however non-factor data lengths will throw an error message as some elements will not be recycled as much as other elements.

```
m1<-matrix(3:5,ncol=3,nrow=2)  #3 is a factor of 6, the matrix size
m1
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    4
## [2,]    4    3    5
```

```
m1<-matrix(3:6,ncol=3,nrow=2)  #4 is not a factor of 6
```

```
## Warning in matrix(3:6, ncol = 3, nrow = 2): data length [4] is not a sub-
## multiple or multiple of the number of columns [3]
```

```
m1
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    3
## [2,]    4    6    4
```

You may have already observed that elements in a matrix are arranged, by default, by columns. You can change this behavior by set the parameter “byrow” as TRUE.

```
m3<-matrix(3:8,ncol = 3,nrow = 2,byrow = TRUE)
m3
```

```
##      [,1] [,2] [,3]
## [1,]    3    4    5
## [2,]    6    7    8
```

It is a good habit to give elements, rows, and columns in R data structure some meaningful names. This will make the maintenance of your data and codes cleaner and easier. This can be done as following:

```
rownames(m3)<-c("Row1","Row2")
colnames(m3)<-c("Col.1","Col.2","Col.3")
m3
```

```
##      Col.1 Col.2 Col.3
## Row1     3     4     5
## Row2     6     7     8
```

```
m3["Row1","Col.2"]
```

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

You can perform all sorts of matrix operations in R. For more details, please refer to https://stats.idre.ucla.edu/r/seminars/r-matrix-operations/ (https://stats.idre.ucla.edu/r/seminars/r-matrix-operations/)

Factor

In R, *factors* are special variables used to store categorical variables. Factor is not an atomic data type and it can be either a numeric data or a character data. There are a few advantages of using factor variable instead of original character (string) variable:

- Factor variables are stored as a vector of integer values, thus it is a more efficient use of memory. Yet, the original set of character values will be used when the factors are displayed, which will make the data presentation more meaningful.
- Many statistical models will automatically handle factor variables properly.
- Factor variables are also very useful in many different types of graphics.

Factors can be defined using *factor()* function. You can use *labels* argument to assign meaningful labels to the values and use *levels* argument to determine the categories of the data. For example,

```
a<-c(0,1,0,0,1)
a.f<-factor(a,labels = c("Male","Female"))
a.f
```

```
## [1] Male   Female Male   Male   Female
## Levels: Male Female
```

You can also directly convert a vector of strings into factors.

```
a<-c("One","Two","Three","One","Three")
a.f<-factor(a)
a.f
```

```
## [1] One   Two   Three One   Three
## Levels: One Three Two
```

```
#However, the levels of the factor are ordered, by default, by alphabet order.
b<-as.integer(a.f)
b
```

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

You can manually assign order of the levels, however, to overcome this problem.

```
a<-c("One","Two","Three","One","Three")
a.f<-factor(a,levels=c("One","Two","Three"))
a.f
```

```
## [1] One   Two   Three One   Three
## Levels: One Two Three
```

```
b<-as.integer(a.f)
b
```

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

Sometimes, you may want to generate factors by specifying the pattern of their levels. *gl()* is a function designed for this purpose. For example,

```
gl(2,8,labels=c("male","female"))
```

```
##  [1] male   male   male   male   male   male   male   male   female female
## [11] female female female female female female
## Levels: male female
```

List

List is more flexible than vector. It allows multiple types of elements, including list itself. A list is constructed by *list()* command. For example,

```
Mike<-list(Name="Mike",Salary=10000,Age=43,Children=c("Tom","Lily","Alice"))
Mike
```

```
## $Name
## [1] "Mike"
##
## $Salary
## [1] 10000
##
## $Age
## [1] 43
##
## $Children
## [1] "Tom"   "Lily"  "Alice"
```

In the above example, we created a profile of Mike in a list structure, where each element is given a name and the elements are of different types. You can retrieve one or multiple elements of a list by either index or name(s). The following examples show various ways of doing so.

```
Mike[2]
```

```
## $Salary
## [1] 10000
```

```
Mike["Salary"]
```

```
## $Salary
## [1] 10000
```

```
Mike$Salary                #$ is a convenient way to retrieve element by element name.
```

```
## [1] 10000
```

```
Mike[c(1,2)]
```

```
## $Name
## [1] "Mike"
##
## $Salary
## [1] 10000
```

```
Mike[c("Salary","Age")]
```

```
## $Salary
## [1] 10000
##
## $Age
## [1] 43
```

str() is a very useful (maybe one of the most frequently used) R function. It compactly display the internal structure of an R object and it is commonly used as a diagnostic function to understand an object. Let's try.

```
str(Mike)
```

```
## List of 4
##  $ Name    : chr "Mike"
##  $ Salary  : num 10000
##  $ Age     : num 43
##  $ Children: chr [1:3] "Tom" "Lily" "Alice"
```

List can contain another list as its element. For example,

```
a<-list(list(1,2),c(3,4))
str(a)
```

```
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : num 2
##  $ : num [1:2] 3 4
```

c() will combine several lists into one. If given a combination of atomic vectors and lists, c() will coerce the vectors to lists before combining them. For example,

```
b<-c(list(1,2),c(3,4))
str(b)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ : num 4
```

Array

Matrix is indeed a special case of Array, which is a multi-dimensional arrangement of data in a vector. An array can be created using *array()* function.

```
a<-array(1:12,c(2,2,3))
a
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

In this course, we seldom use array. If you are interested to know more about array, please do some research on your own.

Basic Data Wrangling

Dr. Liu Qizhang

15 Jan 2019

Introduction

This file gives you an introduction to the Data Frame, and the fundamentals of data wrangling, which includes both data pre-processing and transformation.

Data Structures

Recall the table below from last week's materials, the Data Frame is last data structure we have yet to cover.

	Homogeneous	**Heterogeneous**
1d	Vector	List
2d	Matrix	Data Frame
nd	Array	

It is one of the most useful data structures we have available to us. Its *heterogenity* and *2d* nature means that its elements can contain different atomic data types for different variables yet have the required dimensionality to have different observations.

Data Frame

The simplest way to create manually create a Data Frame is to use the *data.frame()* function and input vectors as parameters.

```
name <- c("Anne", "Pete", "Frank", "Julia", "Cath")
age <- c(28, 30, 21 ,39, 35)
child <- c(FALSE, TRUE, TRUE, FALSE, TRUE)

df <- data.frame(name, age, child)
```

For further exploration and examples use *?data.frame()* to quickly access the documentation.

Note: elements in the same column should be of the same data type.

Data Frame Naming

Just like it was introduced in last week's materials, naming in data frames is no different then in other data structures. Naming can be done after the data structure has been created with the *names()* function:

```
names(df) <- c("Name", "Age", "Child")
df
```

```
##    Name Age Child
## 1  Anne  28 FALSE
## 2  Pete  30  TRUE
## 3 Frank  21  TRUE
## 4 Julia  39 FALSE
## 5  Cath  35  TRUE
```

Or it can be done within the respective data structure's function as the data structure is being created:

```
df <- data.frame("Name" = name, "Age" = age, "Child" = child)
df
```

```
##    Name Age Child
## 1  Anne  28 FALSE
## 2  Pete  30  TRUE
## 3 Frank  21  TRUE
## 4 Julia  39 FALSE
## 5  Cath  35  TRUE
```

```
df <- data.frame(Name = name, Age = age, Child = child)
df
```

```
##    Name Age Child
## 1  Anne  28 FALSE
## 2  Pete  30  TRUE
## 3 Frank  21  TRUE
## 4 Julia  39 FALSE
## 5  Cath  35  TRUE
```

Data Frame Structure

The Data Frame in R is implemented as a *list of vectors* with an important restriction of *equal length vectors*.

```
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ Name : Factor w/ 5 levels "Anne","Cath",..: 1 5 3 4 2
##  $ Age  : num  28 30 21 39 35
##  $ Child: logi  FALSE TRUE TRUE FALSE TRUE
```

R stores the character data type as a factor instead, as it is more space efficient in memory and also because many statistical models we will apply to Data Frames can handle factors automatically.

If you took the time to take a look at the *data.frame()* function's documentation earlier, you may have noticed the *stringsAsFactors* keyword argument, which prevents R from converting the characters to vectors.

```
df <- data.frame(name, age, child,
                 stringsAsFactors = FALSE)
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ name : chr  "Anne" "Pete" "Frank" "Julia" ...
##  $ age  : num  28 30 21 39 35
##  $ child: logi  FALSE TRUE TRUE FALSE TRUE
```

The default, that is when you don't specify this kwarg is equivalent to having *stringsAsFactors = TRUE* as a parameter.

```
df <- data.frame(name, age, child,
                 stringsAsFactors = TRUE)
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ name : Factor w/ 5 levels "Anne","Cath",..: 1 5 3 4 2
##  $ age  : num  28 30 21 39 35
##  $ child: logi  FALSE TRUE TRUE FALSE TRUE
```

Data Frame Subsetting

Data Frame subsetting makes use of both vector and matrix style subsetting with "[]" square brackets, and also the "[[]]" double square brackets and "$" from lists.

```
df
```

```
##    name age child
## 1  Anne  28 FALSE
## 2  Pete  30  TRUE
## 3 Frank  21  TRUE
## 4 Julia  39 FALSE
## 5  Cath  35  TRUE
```

```
df[3,2]       #Selecting a single element with the index
```

```
## [1] 21
```

```
df[3,"age"]   #Selecting a single element with the column name
```

```
## [1] 21
```

```
df[3,]       #Selecting the entire row (Frank's info)
```

```
##    name age child
## 3 Frank  21  TRUE
```

```
df[,2]       #Selecting the entire column with the index (Age Column)
```

```
## [1] 28 30 21 39 35
```

```
df[,"age"]    #Selecting the entire column with the column name (Age Column)
```

```
## [1] 28 30 21 39 35
```

The *c()* vector function can also be used to subset multiple portions of the Data Frame.

```
df[c(3,5), c(2,3)]            #Selecting multiple elements with index
```

```
##   age child
## 3  21  TRUE
## 5  35  TRUE
```

```
df[c(3,5), c("age", "child")]  #Selecting multiple elements with column name
```

```
##   age child
## 3  21  TRUE
## 5  35  TRUE
```

Specifying a subset of a dataframe with only one dimension (using "[]" square brackets) returns a Data Frame and **not a vector**.

```
ages <- df[2]
ages
```

```
##   age
## 1  28
## 2  30
## 3  21
## 4  39
## 5  35
```

```
str(ages)            #a Data Frame that is a list of only one vector
```

```
## 'data.frame':    5 obs. of  1 variable:
##  $ age: num  28 30 21 39 35
```

However, using list style subsetting will instead return a list data structure. Both are valid ways to subset a Data Frame, but return different data structures, so carefully choose which style is appropriate.

```
ages2 <- df$age
ages2
```

```
## [1] 28 30 21 39 35
```

```
str(ages2)           #a list
```

```
##  num [1:5] 28 30 21 39 35
```

```
ages2 <- df[["age"]]  #using column name
ages2
```

```
## [1] 28 30 21 39 35
```

```
str(ages2)           #still a list
```

```
##  num [1:5] 28 30 21 39 35
```

```
ages2 <- df[[2]]      #using index
ages2
```

```
## [1] 28 30 21 39 35
```

```
str(ages2)           #also still a list
```

```
##  num [1:5] 28 30 21 39 35
```

Data Frame Extension

Useful when adding new variables or observations to an existing Data Frame.

Extending by Columns

To add new variables to an existing data frame either use a vector and assign it with list syle subsetting syntax shown earlier

```
height <- c(163, 177, 163, 162, 157)
df$height <- height
df
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
```

```
df[["height"]] <- height
df
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
```

or use the *cbind()* horizontal concatenation function from matrix operations (https://stats.idre.ucla.edu/r/seminars/r-matrix-operations/).

```
weight <- c(74, 63, 68, 55, 56)
cbind(df, weight)                #column bind
```

```
##    name age child height weight
## 1  Anne  28 FALSE    163     74
## 2  Pete  30  TRUE    177     63
## 3 Frank  21  TRUE    163     68
## 4 Julia  39 FALSE    162     55
## 5  Cath  35  TRUE    157     56
```

Extending by Row

As vectors are homogenous, a mini Data Frame can instead be used, before using the *rbind()* function.

```
tom <- data.frame(name = "Tom", age = 37,
                  child = FALSE, height = 183)
rbind(df, tom)                                  #row bind
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
## 6   Tom  37 FALSE    183
```

Data Frame Sorting

Use the *order()* function which returns a vector of sorted index order, instead of using the *sort()* function which returns a vector of sorted elements.

```
sort(df$age)
```

```
## [1] 21 28 30 35 39
```

```
df
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
```

```
ranks <- order(df$age)
ranks                   #the order of the current indexes if sorted
```

```
## [1] 3 1 2 5 4
```

```
df[ranks,]
```

```
##    name age child height
## 3 Frank  21  TRUE    163
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 5  Cath  35  TRUE    157
## 4 Julia  39 FALSE    162
```

Other useful sorting related functions

```
max(df$age)                 #getting the highest age
```

```
## [1] 39
```

```
i_max <- which.max(df$age)
i_max                       #index of the oldest person
```

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

```
df[i_max,1]                 #name of the oldest person
```

```
## [1] Julia
## Levels: Anne Cath Frank Julia Pete
```

```
df$name[i_max]              #alternative way
```

```
## [1] Julia
## Levels: Anne Cath Frank Julia Pete
```

```
min(df$age)                 #getting the lowest age
```

```
## [1] 21
```

Note that the *order()* function is different from the *rank()* function, which only gives us the current ranks of each element.

```
df$age
```

```
## [1] 28 30 21 39 35
```

```
order(df$age)
```

```
## [1] 3 1 2 5 4
```

```
rank(df$age)
```

```
## [1] 2 3 1 5 4
```

Data Frame Indexing

We can use logical operators to find specific cases in our Data Frame. For example, people taller than 171 cm, the average male height in Singapore.

```
df
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
```

```
index <- df$height > 171
index
```

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

```
sum(index)               #number of people taller than the male average
```

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

```
df$name[index]           #person who is taller: pete
```

```
## [1] Pete
## Levels: Anne Cath Frank Julia Pete
```

We can also use the logical operators introduced last week to complete more complex logical operation, such as finding those older than 30 without children.

```
index <- df$age > 30 & df$child == FALSE
index
```

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

```
sum(index)
```

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

```
df$name[index]
```

```
## [1] Julia
## Levels: Anne Cath Frank Julia Pete
```

Other useful indexing related functions

The *which()* function preserves the relevant indexes from a vector or from a logical operation. Useful for keeping index as a smaller variable in memory.

```
x <- c(FALSE, TRUE, FALSE, TRUE, TRUE, FALSE)
which(x)
```

```
## [1] 2 4 5
```

```
index <- which(df$name == "Cath")             #Cath is in the fifth row of the Data Frame
index
```

```
## [1] 5
```

The *match()* function gets all the indexes for elements in a vector, useful when comparing a few different things.

Here we want to get the indexes of Anne, Julia and Cath.

```
index <- match(c("Anne", "Julia", "Cath"), df$name)
index
```

```
## [1] 1 4 5
```

```
df$height[index]
```

```
## [1] 163 162 157
```

The *%in%* function helps us check if elements are in a vector.

```
df
```

```
##    name age child height
## 1  Anne  28 FALSE    163
## 2  Pete  30  TRUE    177
## 3 Frank  21  TRUE    163
## 4 Julia  39 FALSE    162
## 5  Cath  35  TRUE    157
```

```
c("Anne", "Julia", "Cath", "Bob") %in% df$name
```

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

Basic Data Wrangling

We will be using the dpylr package. However, in order to use packages in R we must first install them. There are two ways to install a package in Rstudio, either with the GUI or manually.

The gui method is easier to remember, simply go to *Packages* > *Install* make sure “Install from:” has repository selected and type in “dplyr” into the “Packages” search bar. Then click “install”.

Note: the install dependencies checkbox should not be unchecked unless you know what you’re doing

The manual method is to type and run *install.packages(“dplyr”)* in either the console pane or the script pane. By default, Rstudio will install from its CRAN repository.

To load the package type and run *library(dplyr)*.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

We are going to create the weight vector again and now extend our Data Frame.

```
weight <- c(74, 63, 68, 55, 56)
df$weight <- weight
df
```

```
##    name age child height weight
## 1  Anne  28 FALSE    163     74
## 2  Pete  30  TRUE    177     63
## 3 Frank  21  TRUE    163     68
## 4 Julia  39 FALSE    162     55
## 5  Cath  35  TRUE    157     56
```

Mutate Function

The *mutate()* function can be used to extend a Data Frame, it is flexible enough to be used on both row and column extensions. The first parameter is the Data Frame, while the subsequent parameters will be for adding or dropping variables (set variable = NULL). Here, we will calculate everyone's BMI.

See how much cleaner it is, as dplyr functions know to search variable names within the scope of the Data Frame in the first parameter.

```
df <- mutate(df, bmi = weight/height^2*10000)
df
```

```
##    name age child height weight      bmi
## 1  Anne  28 FALSE    163     74 27.85201
## 2  Pete  30  TRUE    177     63 20.10916
## 3 Frank  21  TRUE    163     68 25.59374
## 4 Julia  39 FALSE    162     55 20.95717
## 5  Cath  35  TRUE    157     56 22.71897
```

To accomplish this without the dplyr package:

```
df$bmi <- df$weight/df$height^2*10000
df
```

```
##    name age child height weight      bmi
## 1  Anne  28 FALSE    163     74 27.85201
## 2  Pete  30  TRUE    177     63 20.10916
## 3 Frank  21  TRUE    163     68 25.59374
## 4 Julia  39 FALSE    162     55 20.95717
## 5  Cath  35  TRUE    157     56 22.71897
```

We can also make our non dplyr method dplyr-like with the *attach()* and *detach() functions which attach and detach a database (in this case a Data Frame) to the R search path.

```
attach(df)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     age, child, height, name, weight
```

```
df$bmi <- weight/height^2*10000
detach(df)                        #remember to detach the database
```

Filter Function

The *filter()* function helps us subset rows. The first variable will be our Data Frame, with the next parameter our logical filter operation. In our example, we want to find people who have a healthy BMI, between 18.5 and 24.9.

```
filter(df, bmi > 18.5 & bmi < 24.9)
```

```
##    name age child height weight      bmi
## 1  Pete  30  TRUE    177     63 20.10916
## 2 Julia  39 FALSE    162     55 20.95717
## 3  Cath  35  TRUE    157     56 22.71897
```

To accomplish this without the dplyr package:

```
df[df$bmi > 18.5 & df$bmi < 24.9,]  #don't forget the "," after as it is row level
```

```
##    name age child height weight      bmi
## 2  Pete  30  TRUE    177     63 20.10916
## 4 Julia  39 FALSE    162     55 20.95717
## 5  Cath  35  TRUE    157     56 22.71897
```

Select Function

The *select()* function helps us subset columns.The first parameter is the Data Frame, while the subsequent parameters the columns we want to select. Notice that in our filter example the Data Frame contains extraneous information not relevant to us, so we will make a new Data Frame with only vital variables.

```
health <- select(df, name, height, weight, bmi)  #it can be argued that age in relation to bm
i is relevant, but for the sake of this example it will not be included
filter(health, bmi > 18.5 & bmi < 24.9)          #much better
```

```
##    name height weight      bmi
## 1  Pete    177     63 20.10916
## 2 Julia    162     55 20.95717
## 3  Cath    157     56 22.71897
```

To accomplish this without the dplyr package:

```
df[,c("name", "height", "weight", "bmi")]  #column level so "," before
```

```
##    name height weight      bmi
## 1  Anne    163     74 27.85201
## 2  Pete    177     63 20.10916
## 3 Frank    163     68 25.59374
## 4 Julia    162     55 20.95717
## 5  Cath    157     56 22.71897
```

```
df[df$bmi > 18.5 & df$bmi < 24.9,c("name", "height", "weight", "bmi")]  #here it is done in o
ne line without dplyr
```

```
##    name height weight      bmi
## 2  Pete    177     63 20.10916
## 4 Julia    162     55 20.95717
## 5  Cath    157     56 22.71897
```

Pipe Operator Function

The %>% pipe operator function helps us chain these three functions together. We will repeat the previous example but now all in one line of code.

```
df %>% select(name, height, weight, bmi) %>% filter(bmi > 18.5 & bmi < 24.9)
```

```
##    name height weight      bmi
## 1  Pete    177     63 20.10916
## 2 Julia    162     55 20.95717
## 3  Cath    157     56 22.71897
```

Note that the use of the Data Frame in the first parameter in select and filter is no longer needed, neither is the intermediate health Data Frame.

SRX Questions

SRX.csv from week two in the overview is required for the following questions.

Firstly, we must import the dataset, rather than create our own as we did earlier. We can import the dataset in two ways:

We can either use the user interface from Rstudio by going into the environment pane > import dataset > from text (readr) > browse… and select SRX.csv. Check that it is the right csv file in the data preview pane. Note the import options and code preview sections. Once ready, press import.

Or, datasets can also be imported with code. Which would be the same code that the user interface executes.

```
library(readr)
SRX_Data <- read_csv("SRX_Data.csv")  #note here my working directory is set to where the csv
is located
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   X1 = col_double(),
##   Built.Year. = col_double(),
##   Bathrooms. = col_double(),
##   No..of.Units. = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

```
View(SRX_Data)                        #to view the dataset, or click on the icon for SRX_Data
in environment pane > Data
```

Try the following questions:

Notice that we have a spare column X1 as our dataset already has an index. How can we easily fix this without using dplyr?

SRX_Data <- SRX_Data[-1]

What is the dplyr solution?

SRX_Data <- select(SRX_Data, -X1)

Notice that using a libraries solution may not always be the shortest or easiest way.

Can this be accomplished as one-liner from importing dataset to dropping column 1? Do this without dplyr and with it.

SRX_Data <- read_csv("SRX_Data.csv")[-1]

SRX_Data <- read_csv("SRX_Data.csv") %>% select(-x1)

How can we check the structure of our data set?

str(SRX_Data)

How many units are there in Laverne's Loft?

SRX_Data[1,15]

Get me all the asking prices. Do it in at least two different ways (there are three ways taught)

SRX_Data[,2] SRX_Data[,"Asking."] select(SRX_Data, Asking.)

What about the built year and the facilities of places that were built after 2000?

SRX_Data[SRX_Data$Built.Year. > 2000 ,c(5,16)] SRX_Data[SRX_Data$Built.Year. > 2000 ,c("Built.Year.","Facilities")] SRX_Data %>% select(Built.Year., Facilities) %>% filter(Built.Year. > 2000)

Try playing around with the dataset with the other useful Data Frame functions or even extend the dataset with your own variables or observations.

Advanced Data Wrangling

Dr. Liu Qizhang

31 January 2019

Introduction

This file further expands on data wrangling, how to import data from various sources into R and also how to deal with tiny data and the various ways we can combine our data that we have just imported.

Importing Data

It is much more common for us to import data rather then to artificially create our own or to manually input it by hand. Therefore, we will cover how to import data with the use of the tidyverse packages (https://www.tidyverse.org/) readr, readxl and tidyr. We can also import data with R-base functions. Notice that we have already made use of readr, and how to install packages in last week's SGX questions when we had to import that data, and that dplyr and tibble are also part of tidyverse with dplyr used explicitly while tibble was used to display our data when using dplyr.

Data is stored in delimiter-separated values (https://en.wikipedia.org/wiki/Delimiter-separated_values), the common ones that we are aware of - excel spreadsheets and google sheets are simply propriety versions of this. You may have seen csv(comma separated values) or tsv(tab-separated values) where commas "," or tabs

are used to partition the cells of data.

Via readr

Here are some of the more common readr functions we will work with, as shown in the video. Readr cheatsheet (https://rawgit.com/rstudio/cheatsheets/master/data-import.pdf)

However, there are still more functions (https://cran.r-project.org/web/packages/readr/readr.pdf) in the documentation which you may explore on your own.

Function	Format	Typical Suffix
read_table	whitespace separated values	txt
read_csv	comma separated values	csv
read_csv2	semicolon separated values	csv
read_tsv	tab delimited separated values	tsv
read_delim	general text file format, must specify delimiter	txt

Checking format

Here we will use Prof Rafael's murders dataset from before, which is from the *dslabs* package. Code below taken from his github (https://rafalab.github.io/dsbook/importing-data.html), copy and run this code to copy and paste this dataset into your current working directory.

```
install.packages("dslabs")
```

```
library(dslabs)
dir <- system.file(package="dslabs") #extracts the location of package
filename <- file.path(dir,"extdata/murders.csv")
file.copy(filename, "murders.csv")
```

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

We use readr's *read_lines()* function to double check if it really is comma separated.

```
library(readr)
read_lines("murders.csv", n_max = 3) #n_max - max number of lines to read
```

```
## [1] "state,abb,region,population,total" "Alabama,AL,South,4779736,135"
## [3] "Alaska,AK,West,710231,19"
```

A visual confirmation of commas and the correct file suffix means that we can use the *read_csv()* function. We can use the fullpath - e.g. c:/user//documents/rsessions/"murders.csv" or, as we previously copied the csv to our current working directory, simply use the filename "murders.csv".

```
dat <- read_csv("murders.csv")
```

```
## Parsed with column specification:
## cols(
##   state = col_character(),
##   abb = col_character(),
##   region = col_character(),
##   population = col_double(),
##   total = col_double()
## )
```

As we used tidyverse packages for data import, displaying our data with the *head()* function display it as a tibble.

```
head(dat)
```

```
## # A tibble: 6 x 5
##   state      abb   region population total
##   <chr>      <chr> <chr>       <dbl> <dbl>
## 1 Alabama    AL    South     4779736   135
## 2 Alaska     AK    West       710231    19
## 3 Arizona    AZ    West      6392017   232
## 4 Arkansas   AR    South     2915918    93
## 5 California CA    West     37253956  1257
## 6 Colorado   CO    West      5029196    65
```

Via readxl

Below are the common readxl functions. The readxl website (https://readxl.tidyverse.org/) contains more examples, I recommend reading through the usage section on the same page.

Function	Format	Typical Suffix
read_excel	auto detects the format	xls, xlsx
read_xls	original format	xls
read_xlsx	new excel format	xlsx

Firstly, install the package as shown last week.

```
install.packages("readxl")
```

```
library(readxl)
```

Here we will use readxl's built in datasets, many libraries have this for practice and example purposes, here datasets.xlsx contains many famous datasets such as the one below.

```
xlsx_example <- readxl_example("datasets.xlsx")
read_excel(xlsx_example)
```

```
## readxl works best with a newer version of the tibble package.
## You currently have tibble v1.4.2.
## Falling back to column name repair from tibble <= v1.4.2.
## Message displays once per session.
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            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           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

To simply display the worksheet names in this work book.

```
excel_sheets(xlsx_example)
```

```
## [1] "iris"     "mtcars"   "chickwts" "quakes"
```

Either specify the sheet number (1-based indexing), or the sheet's name

```
read_excel(xlsx_example, sheet = "chickwts")
```

```
## # A tibble: 71 x 2
##    weight feed
##     <dbl> <chr>
##  1    179 horsebean
##  2    160 horsebean
##  3    136 horsebean
##  4    227 horsebean
##  5    217 horsebean
##  6    168 horsebean
##  7    108 horsebean
##  8    124 horsebean
##  9    143 horsebean
## 10    140 horsebean
## # ... with 61 more rows
```

```
read_excel(xlsx_example, sheet = 3)
```

```
## # A tibble: 71 x 2
##    weight feed
##     <dbl> <chr>
##  1    179 horsebean
##  2    160 horsebean
##  3    136 horsebean
##  4    227 horsebean
##  5    217 horsebean
##  6    168 horsebean
##  7    108 horsebean
##  8    124 horsebean
##  9    143 horsebean
## 10    140 horsebean
## # ... with 61 more rows
```

Via R-base

R-base functions such as *read.csv()* and *read.table()* can be used without having to install any libraries or having to rely on our IDE's (RStudio) functionality.

Notice that as the tibble package displays data imported using tidyverse packages such as readr and readxl, our data is in the data.frame form instead of as a tibble.

```
dat2 <- read.csv("murders.csv")
class(dat2)
```

```
## [1] "data.frame"
```

As we mentioned before, R is special in that it has the *factor* special variable which many statistical models in R can handle so this R-base import function will automatically convert any character strings to factors. In order to avoid this, we can use the *stringsAsFactors* keyword argument to switch off this behaviour.

```
class(dat2$abb)
```

```
## [1] "factor"
```

```
class(dat2$region)
```

```
## [1] "factor"
```

```
dat3 <- read.csv("murders.csv", stringsAsFactors = FALSE)
class(dat3$abb)
```

```
## [1] "character"
```

```
class(dat2$region)
```

```
## [1] "factor"
```

CSV Example

In this course we will heavily rely on the CSV file format. It has widespread use in the data science community due to its efficiency at storing large amounts of data and also as it is platform agnostic.There is also no size limit with csv files.

```
care.data <- read.csv("hospital-data.csv")
str(care.data)
```

```
## 'data.frame':    4826 obs. of  13 variables:
##  $ Provider.Number   : Factor w/ 4826 levels "010001","010005",..: 1 2 3 4 5 6 7 8 9 10
...
##  $ Hospital.Name     : Factor w/ 4624 levels "ABBEVILLE AREA MEDICAL CENTER",..: 3674 2203
1068 2564 857 1482 2202 3918 929 3699 ...
##  $ Address.1         : Factor w/ 4800 levels " 1200 EAST PECAN ST",..: 459 2103 1726 3926
228 1681 4182 3224 1582 2642 ...
##  $ Address.2         : logi  NA NA NA NA NA NA ...
##  $ Address.3         : logi  NA NA NA NA NA NA ...
##  $ City              : Factor w/ 3018 levels "ABBEVILLE","ABERDEEN",..: 723 278 899 1973 1
560 1140 1099 250 929 2679 ...
##  $ State             : Factor w/ 56 levels "AK","AL","AR",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ ZIP.Code          : int  36301 35957 35631 36467 36049 35640 35976 35235 35968 36784
...
##  $ County            : Factor w/ 1533 levels "","ABBEVILLE",..: 652 861 785 347 356 939 86
1 695 388 297 ...
##  $ Phone.Number      : num  3.35e+09 2.57e+09 2.57e+09 3.34e+09 3.34e+09 ...
##  $ Hospital.Type     : Factor w/ 4 levels "ACUTE CARE - VETERANS ADMINISTRATION",..: 2 2 2
2 2 2 2 2 2 2 ...
##  $ Hospital.Ownership: Factor w/ 14 levels "Government-Federal",..: 5 5 5 14 9 9 5 14 9 9
...
##  $ Emergency.Services: Factor w/ 3 levels "No","Not Available",..: 3 3 3 3 3 2 3 3 3 2 ...
```

Because of the lack of file size limit, sometimes we might want to be smart about how we import data so as to save on memory space - and as a result on time taken.

While the time differences here are small due to the filesize, this can be handy when our files are 100s of gbs large.

```
system.time(care.data<-read.csv("hospital-data.csv"))
```

```
##    user  system elapsed
##    0.08    0.02    0.09
```

```
data.sample <- read.csv("hospital-data.csv", nrows = 5)
data.sample
```

```
##   Provider.Number                    Hospital.Name
## 1           10001 SOUTHEAST ALABAMA MEDICAL CENTER
## 2           10005    MARSHALL MEDICAL CENTER SOUTH
## 3           10006   ELIZA COFFEE MEMORIAL HOSPITAL
## 4           10007          MIZELL MEMORIAL HOSPITAL
## 5           10008      CRENSHAW COMMUNITY HOSPITAL
##                     Address.1 Address.2 Address.3     City State ZIP.Code
## 1     1108 ROSS CLARK CIRCLE        NA        NA   DOTHAN    AL    36301
## 2 2505 U S HIGHWAY 431 NORTH        NA        NA     BOAZ    AL    35957
## 3         205 MARENGO STREET        NA        NA FLORENCE    AL    35631
## 4              702 N MAIN ST        NA        NA      OPP    AL    36467
## 5        101 HOSPITAL CIRCLE        NA        NA  LUVERNE    AL    36049
##       County Phone.Number        Hospital.Type
## 1    HOUSTON   3347938701 Acute Care Hospitals
## 2   MARSHALL   2565938310 Acute Care Hospitals
## 3 LAUDERDALE   2567688400 Acute Care Hospitals
## 4  COVINGTON   3344933541 Acute Care Hospitals
## 5   CRENSHAW   3343353374 Acute Care Hospitals
##                           Hospital.Ownership Emergency.Services
## 1 Government - Hospital District or Authority               Yes
## 2 Government - Hospital District or Authority               Yes
## 3 Government - Hospital District or Authority               Yes
## 4              Voluntary non-profit - Private               Yes
## 5                             Proprietary               Yes
```

```
colclass <-c("character","character","character",
             "character","character","character",
             "factor", "factor", "factor",
             "character", "factor", "factor", "factor")
system.time(care.data<-read.csv("hospital-data.csv", colClasses = colclass))
```

```
##    user  system elapsed
##    0.05    0.00    0.04
```

Via URL

Very often, we will obtain our datasets from an online source. Instead of first downloading them to be imported, we can skip a step and read them directly from the URL. In this case we will be using readr's *read_csv()* function.

```
url <- "https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"
dat <- read_csv(url)
```

```
## Parsed with column specification:
## cols(
##   state = col_character(),
##   abb = col_character(),
##   region = col_character(),
##   population = col_double(),
##   total = col_double()
## )
```

There are also some useful functions associated with of importing data via URL.

Sometimes, you might also want to keep a local copy of the file.

```
download.file(url, "murders.csv")
```

Additionally, it is useful to have a temporary directory or filename auto generated to manage these URL imports. We can do this with the *tempdir()* and *tempfile()* functions respectively.

```
tempdir()
```

```
## [1] "C:\\Users\\bizlqz\\AppData\\Local\\Temp\\RtmpYXa3Ky"
```

```
tempfile()
```

```
## [1] "C:\\Users\\bizlqz\\AppData\\Local\\Temp\\RtmpYXa3Ky\\file15585f212826"
```

Here is an example of how we might use these functions.

```
tmp_filename <- tempfile()
download.file(url, tmp_filename)
dat <- read_csv(tmp_filename)
```

```
## Parsed with column specification:
## cols(
##   state = col_character(),
##   abb = col_character(),
##   region = col_character(),
##   population = col_double(),
##   total = col_double()
## )
```

```
file.remove(tmp_filename)
```

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

There are many places on the internet to source for these datasets, such as kaggle (www.kaggle.com), kdnuggets (www.kdnuggets.com/dataset/index.html). For sg related open data (https://en.wikipedia.org/wiki/Open_data) visit data.gov.sg (https://data.gov.sg/). There are also european (https://data.europa.eu/euodp/en/home) and US (https://www.data.gov/) counterparts.

Via JSON

Sometimes open data is provided via API(Application Programming Interface), especially in cases where the data is dynamic. The interfaces make use of JSON (https://www.json.org/), JavaScript Object Notation, which is a lightweight language independent standard used to exchange data between programs.

```
{
  "firstName": "Lilian",
  "lastName": "Teo",
  "age": 42,
  "spouse": null
}
```

Notice how the data is nested in JSON format.

```
{
  "firstName": "Lilian",
  "lastName": "Teo",
  "age": 42,
  "spouse": null,
  "children":[
    {
      "Name":"Melvyn",
      "age": 12
    },
    {
      "Name":"Joan",
      "age": 8
    }
    }
  ]
}
```

An example of how to import JSON files, combining our knowledge of importing from URL above.

```
library(jsonlite)
url <- "https://api.data.gov.sg/v1/transport/carpark-availability"
data <- fromJSON(url)
a <- as.data.frame(data$items$carpark_data)
```

Via XML

When we are obtaining our data from crawling a website, we will make use of XML (https://en.wikipedia.org/wiki/XML) - eXtensible Markup Language, which is how websites store and transport data.

Here is how our JSON from before would look in XML. Notice how data is enclosed by starting and ending tags.

```
<?xml version="1.0"?>
<person>
  <firstName>Lilian</firstName>
  <lastName>Teo</lastName>
  <age>32</age>
  <spouse></spouse>
</person>
```

Here is how the second example would look. Notice that XML also has a tree structure with the corresponding tags being nodes in the tree where the root node of the three has five child nodes.

```
<?xml version="1.0"?>
<person>
  <firstName>Lilian</firstName> <!--the data inside is called XML text-->
  <lastName>Teo</lastName>
  <age>32</age>
  <spouse></spouse>
  <children>
         <child gender="male">  <!--notice how these tags can have attributes-->
                 <name>Melvyn</name>
                 <age>12</age>
         </child>
         <child gender="female">
                 <name>Joan</name>
                 <age>8</age>
         </child>
  <children>
</person>
```

XML Example

XML can store all types of data while HTML text must be pre-defined. When importing XML remember to install the *XML package*. Use the *xmlParse()* function to import the data from “books.xml”.

```
library(XML)
data <- xmlParse("books.xml")
```

Here is how the data should look.

```
data
```

```
## <?xml version="1.0"?>
## <library>
##   <location>HSSML</location>
##   <catalog>
##     <book id="bk101" type="HardCover">
##       <author>Gambardella, Matthew</author>
##       <title>XML Developer's Guide</title>
##       <genre>Computer</genre>
##       <price>44.95</price>
##       <publish_date>2000-10-01</publish_date>
##       <description>An in-depth look at creating applications
##         with XML.</description>
##     </book>
##     <book id="bk102" type="HardCover">
##       <author>Ralls, Kim</author>
##       <title>Midnight Rain</title>
##       <genre>Fantasy</genre>
##       <price>5.95</price>
##       <publish_date>2000-12-16</publish_date>
##       <description>A former architect battles corporate zombies,
##         an evil sorceress, and her own childhood to become queen
##         of the world.</description>
##     </book>
##     <book id="bk103" type="HardCover">
##       <author>Corets, Eva</author>
##       <title>Maeve Ascendant</title>
##       <genre>Fantasy</genre>
##       <price>5.95</price>
##       <publish_date>2000-11-17</publish_date>
##       <description>After the collapse of a nanotechnology
##         society in England, the young survivors lay the
##         foundation for a new society.</description>
##     </book>
##     <book id="bk104" type="SoftCover">
##       <author>Corets, Eva</author>
##       <title>Oberon's Legacy</title>
##       <genre>Fantasy</genre>
##       <price>5.95</price>
##       <publish_date>2001-03-10</publish_date>
##       <description>In post-apocalypse England, the mysterious
##         agent known only as Oberon helps to create a new life
##         for the inhabitants of London. Sequel to Maeve
##         Ascendant.</description>
##     </book>
##     <book id="bk105" type="HardCover">
##       <author>Corets, Eva</author>
##       <title>The Sundered Grail</title>
##       <genre>Fantasy</genre>
##       <price>5.95</price>
##       <publish_date>2001-09-10</publish_date>
##       <description>The two daughters of Maeve, half-sisters,
##         battle one another for control of England. Sequel to
##         Oberon's Legacy.</description>
##     </book>
##     <book id="bk106" type="SoftCover">
##       <author>Randall, Cynthia</author>
##       <title>Lover Birds</title>
```

```
##       <genre>Romance</genre>
##       <price>4.95</price>
##       <publish_date>2000-09-02</publish_date>
##       <description>When Carla meets Paul at an ornithology
##         conference, tempers fly as feathers get ruffled.</description>
##     </book>
##     <book id="bk107" type="HardCover">
##       <author>Thurman, Paula</author>
##       <title>Splish Splash</title>
##       <genre>Romance</genre>
##       <price>4.95</price>
##       <publish_date>2000-11-02</publish_date>
##       <description>A deep sea diver finds true love twenty
##         thousand leagues beneath the sea.</description>
##     </book>
##     <book id="bk108" type="SoftCover">
##       <author>Knorr, Stefan</author>
##       <title>Creepy Crawlies</title>
##       <genre>Horror</genre>
##       <price>4.95</price>
##       <publish_date>2000-12-06</publish_date>
##       <description>An anthology of horror stories about roaches,
##         centipedes, scorpions  and other insects.</description>
##     </book>
##     <book id="bk109" type="SoftCover">
##       <author>Kress, Peter</author>
##       <title>Paradox Lost</title>
##       <genre>Science Fiction</genre>
##       <price>6.95</price>
##       <publish_date>2000-11-02</publish_date>
##       <description>After an inadvertant trip through a Heisenberg
##         Uncertainty Device, James Salway discovers the problems
##         of being quantum.</description>
##     </book>
##     <book id="bk110" type="HardCover">
##       <author>O'Brien, Tim</author>
##       <title>Microsoft .NET: The Programming Bible</title>
##       <genre>Computer</genre>
##       <price>36.95</price>
##       <publish_date>2000-12-09</publish_date>
##       <description>Microsoft's .NET initiative is explored in
##         detail in this deep programmer's reference.</description>
##     </book>
##     <book id="bk111" type="HardCover">
##       <author>O'Brien, Tim</author>
##       <title>MSXML3: A Comprehensive Guide</title>
##       <genre>Computer</genre>
##       <price>36.95</price>
##       <publish_date>2000-12-01</publish_date>
##       <description>The Microsoft MSXML3 parser is covered in
##         detail, with attention to XML DOM interfaces, XSLT processing,
##         SAX and more.</description>
##     </book>
##     <book id="bk112" type="SoftCover">
##       <author>Galos, Mike</author>
##       <title>Visual Studio 7: A Comprehensive Guide</title>
##       <genre>Computer</genre>
##       <price>49.95</price>
```

```
##       <publish_date>2001-04-16</publish_date>
##       <description>Microsoft Visual Studio 7 is explored in depth,
##         looking at how Visual Basic, Visual C++, C#, and ASP+ are
##         integrated into a comprehensive development
##         environment.</description>
##     </book>
##   </catalog>
## </library>
##
```

Now we use the *xmlRoot()* function to access the root node of the tree.

```
root <- xmlRoot(data)
```

xmlChildren() function gets use the children nodes of the tree.

```
nodes <- xmlChildren(root) #notice the input here should be root not data
```

Below is the second child node of root, however to go even further we can get the child node of this node.

```
a <- nodes[[2]]
a
```

```
## <catalog>
##   <book id="bk101" type="HardCover">
##     <author>Gambardella, Matthew</author>
##     <title>XML Developer's Guide</title>
##     <genre>Computer</genre>
##     <price>44.95</price>
##     <publish_date>2000-10-01</publish_date>
##     <description>An in-depth look at creating applications
##         with XML.</description>
##   </book>
##   <book id="bk102" type="HardCover">
##     <author>Ralls, Kim</author>
##     <title>Midnight Rain</title>
##     <genre>Fantasy</genre>
##     <price>5.95</price>
##     <publish_date>2000-12-16</publish_date>
##     <description>A former architect battles corporate zombies,
##         an evil sorceress, and her own childhood to become queen
##         of the world.</description>
##   </book>
##   <book id="bk103" type="HardCover">
##     <author>Corets, Eva</author>
##     <title>Maeve Ascendant</title>
##     <genre>Fantasy</genre>
##     <price>5.95</price>
##     <publish_date>2000-11-17</publish_date>
##     <description>After the collapse of a nanotechnology
##         society in England, the young survivors lay the
##         foundation for a new society.</description>
##   </book>
##   <book id="bk104" type="SoftCover">
##     <author>Corets, Eva</author>
##     <title>Oberon's Legacy</title>
##     <genre>Fantasy</genre>
##     <price>5.95</price>
##     <publish_date>2001-03-10</publish_date>
##     <description>In post-apocalypse England, the mysterious
##         agent known only as Oberon helps to create a new life
##         for the inhabitants of London. Sequel to Maeve
##         Ascendant.</description>
##   </book>
##   <book id="bk105" type="HardCover">
##     <author>Corets, Eva</author>
##     <title>The Sundered Grail</title>
##     <genre>Fantasy</genre>
##     <price>5.95</price>
##     <publish_date>2001-09-10</publish_date>
##     <description>The two daughters of Maeve, half-sisters,
##         battle one another for control of England. Sequel to
##         Oberon's Legacy.</description>
##   </book>
##   <book id="bk106" type="SoftCover">
##     <author>Randall, Cynthia</author>
##     <title>Lover Birds</title>
##     <genre>Romance</genre>
##     <price>4.95</price>
##     <publish_date>2000-09-02</publish_date>
```

```
##     <description>When Carla meets Paul at an ornithology
##         conference, tempers fly as feathers get ruffled.</description>
##   </book>
##   <book id="bk107" type="HardCover">
##     <author>Thurman, Paula</author>
##     <title>Splish Splash</title>
##     <genre>Romance</genre>
##     <price>4.95</price>
##     <publish_date>2000-11-02</publish_date>
##     <description>A deep sea diver finds true love twenty
##         thousand leagues beneath the sea.</description>
##   </book>
##   <book id="bk108" type="SoftCover">
##     <author>Knorr, Stefan</author>
##     <title>Creepy Crawlies</title>
##     <genre>Horror</genre>
##     <price>4.95</price>
##     <publish_date>2000-12-06</publish_date>
##     <description>An anthology of horror stories about roaches,
##         centipedes, scorpions  and other insects.</description>
##   </book>
##   <book id="bk109" type="SoftCover">
##     <author>Kress, Peter</author>
##     <title>Paradox Lost</title>
##     <genre>Science Fiction</genre>
##     <price>6.95</price>
##     <publish_date>2000-11-02</publish_date>
##     <description>After an inadvertant trip through a Heisenberg
##         Uncertainty Device, James Salway discovers the problems
##         of being quantum.</description>
##   </book>
##   <book id="bk110" type="HardCover">
##     <author>O'Brien, Tim</author>
##     <title>Microsoft .NET: The Programming Bible</title>
##     <genre>Computer</genre>
##     <price>36.95</price>
##     <publish_date>2000-12-09</publish_date>
##     <description>Microsoft's .NET initiative is explored in
##         detail in this deep programmer's reference.</description>
##   </book>
##   <book id="bk111" type="HardCover">
##     <author>O'Brien, Tim</author>
##     <title>MSXML3: A Comprehensive Guide</title>
##     <genre>Computer</genre>
##     <price>36.95</price>
##     <publish_date>2000-12-01</publish_date>
##     <description>The Microsoft MSXML3 parser is covered in
##         detail, with attention to XML DOM interfaces, XSLT processing,
##         SAX and more.</description>
##   </book>
##   <book id="bk112" type="SoftCover">
##     <author>Galos, Mike</author>
##     <title>Visual Studio 7: A Comprehensive Guide</title>
##     <genre>Computer</genre>
##     <price>49.95</price>
##     <publish_date>2001-04-16</publish_date>
##     <description>Microsoft Visual Studio 7 is explored in depth,
##         looking at how Visual Basic, Visual C++, C#, and ASP+ are
```

```
##         integrated into a comprehensive development
##         environment.</description>
##   </book>
## </catalog>
```

```
moreNodes <- xmlChildren(a)
b <- moreNodes[[2]]
b
```

```
## <book id="bk102" type="HardCover">
##   <author>Ralls, Kim</author>
##   <title>Midnight Rain</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2000-12-16</publish_date>
##   <description>A former architect battles corporate zombies,
##         an evil sorceress, and her own childhood to become queen
##         of the world.</description>
## </book>
```

If we knbw the structure of the XML file we can use tags or attributes to search for specific data.Here is how we just get the books.

```
books <- getNodeSet(data,"/library/catalog/book")
books
```

```
## [[1]]
## <book id="bk101" type="HardCover">
##   <author>Gambardella, Matthew</author>
##   <title>XML Developer's Guide</title>
##   <genre>Computer</genre>
##   <price>44.95</price>
##   <publish_date>2000-10-01</publish_date>
##   <description>An in-depth look at creating applications
##         with XML.</description>
## </book>
##
## [[2]]
## <book id="bk102" type="HardCover">
##   <author>Ralls, Kim</author>
##   <title>Midnight Rain</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2000-12-16</publish_date>
##   <description>A former architect battles corporate zombies,
##         an evil sorceress, and her own childhood to become queen
##         of the world.</description>
## </book>
##
## [[3]]
## <book id="bk103" type="HardCover">
##   <author>Corets, Eva</author>
##   <title>Maeve Ascendant</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2000-11-17</publish_date>
##   <description>After the collapse of a nanotechnology
##         society in England, the young survivors lay the
##         foundation for a new society.</description>
## </book>
##
## [[4]]
## <book id="bk104" type="SoftCover">
##   <author>Corets, Eva</author>
##   <title>Oberon's Legacy</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2001-03-10</publish_date>
##   <description>In post-apocalypse England, the mysterious
##         agent known only as Oberon helps to create a new life
##         for the inhabitants of London. Sequel to Maeve
##         Ascendant.</description>
## </book>
##
## [[5]]
## <book id="bk105" type="HardCover">
##   <author>Corets, Eva</author>
##   <title>The Sundered Grail</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2001-09-10</publish_date>
##   <description>The two daughters of Maeve, half-sisters,
##         battle one another for control of England. Sequel to
```

```
##         Oberon's Legacy.</description>
## </book>
##
## [[6]]
## <book id="bk106" type="SoftCover">
##   <author>Randall, Cynthia</author>
##   <title>Lover Birds</title>
##   <genre>Romance</genre>
##   <price>4.95</price>
##   <publish_date>2000-09-02</publish_date>
##   <description>When Carla meets Paul at an ornithology
##         conference, tempers fly as feathers get ruffled.</description>
## </book>
##
## [[7]]
## <book id="bk107" type="HardCover">
##   <author>Thurman, Paula</author>
##   <title>Splish Splash</title>
##   <genre>Romance</genre>
##   <price>4.95</price>
##   <publish_date>2000-11-02</publish_date>
##   <description>A deep sea diver finds true love twenty
##         thousand leagues beneath the sea.</description>
## </book>
##
## [[8]]
## <book id="bk108" type="SoftCover">
##   <author>Knorr, Stefan</author>
##   <title>Creepy Crawlies</title>
##   <genre>Horror</genre>
##   <price>4.95</price>
##   <publish_date>2000-12-06</publish_date>
##   <description>An anthology of horror stories about roaches,
##         centipedes, scorpions  and other insects.</description>
## </book>
##
## [[9]]
## <book id="bk109" type="SoftCover">
##   <author>Kress, Peter</author>
##   <title>Paradox Lost</title>
##   <genre>Science Fiction</genre>
##   <price>6.95</price>
##   <publish_date>2000-11-02</publish_date>
##   <description>After an inadvertant trip through a Heisenberg
##         Uncertainty Device, James Salway discovers the problems
##         of being quantum.</description>
## </book>
##
## [[10]]
## <book id="bk110" type="HardCover">
##   <author>O'Brien, Tim</author>
##   <title>Microsoft .NET: The Programming Bible</title>
##   <genre>Computer</genre>
##   <price>36.95</price>
##   <publish_date>2000-12-09</publish_date>
##   <description>Microsoft's .NET initiative is explored in
##         detail in this deep programmer's reference.</description>
## </book>
```

```
##
## [[11]]
## <book id="bk111" type="HardCover">
##   <author>O'Brien, Tim</author>
##   <title>MSXML3: A Comprehensive Guide</title>
##   <genre>Computer</genre>
##   <price>36.95</price>
##   <publish_date>2000-12-01</publish_date>
##   <description>The Microsoft MSXML3 parser is covered in
##         detail, with attention to XML DOM interfaces, XSLT processing,
##         SAX and more.</description>
## </book>
##
## [[12]]
## <book id="bk112" type="SoftCover">
##   <author>Galos, Mike</author>
##   <title>Visual Studio 7: A Comprehensive Guide</title>
##   <genre>Computer</genre>
##   <price>49.95</price>
##   <publish_date>2001-04-16</publish_date>
##   <description>Microsoft Visual Studio 7 is explored in depth,
##         looking at how Visual Basic, Visual C++, C#, and ASP+ are
##         integrated into a comprehensive development
##         environment.</description>
## </book>
##
## attr(,"class")
## [1] "XMLNodeSet"
```

We can further refine our search for just hard cover books.

```
books <- getNodeSet(data, "/library/catalog/book[@type='HardCover']")
books
```

```
## [[1]]
## <book id="bk101" type="HardCover">
##   <author>Gambardella, Matthew</author>
##   <title>XML Developer's Guide</title>
##   <genre>Computer</genre>
##   <price>44.95</price>
##   <publish_date>2000-10-01</publish_date>
##   <description>An in-depth look at creating applications
##         with XML.</description>
## </book>
##
## [[2]]
## <book id="bk102" type="HardCover">
##   <author>Ralls, Kim</author>
##   <title>Midnight Rain</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2000-12-16</publish_date>
##   <description>A former architect battles corporate zombies,
##         an evil sorceress, and her own childhood to become queen
##         of the world.</description>
## </book>
##
## [[3]]
## <book id="bk103" type="HardCover">
##   <author>Corets, Eva</author>
##   <title>Maeve Ascendant</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2000-11-17</publish_date>
##   <description>After the collapse of a nanotechnology
##         society in England, the young survivors lay the
##         foundation for a new society.</description>
## </book>
##
## [[4]]
## <book id="bk105" type="HardCover">
##   <author>Corets, Eva</author>
##   <title>The Sundered Grail</title>
##   <genre>Fantasy</genre>
##   <price>5.95</price>
##   <publish_date>2001-09-10</publish_date>
##   <description>The two daughters of Maeve, half-sisters,
##         battle one another for control of England. Sequel to
##         Oberon's Legacy.</description>
## </book>
##
## [[5]]
## <book id="bk107" type="HardCover">
##   <author>Thurman, Paula</author>
##   <title>Splish Splash</title>
##   <genre>Romance</genre>
##   <price>4.95</price>
##   <publish_date>2000-11-02</publish_date>
##   <description>A deep sea diver finds true love twenty
##         thousand leagues beneath the sea.</description>
## </book>
```

```
##
## [[6]]
## <book id="bk110" type="HardCover">
##   <author>O'Brien, Tim</author>
##   <title>Microsoft .NET: The Programming Bible</title>
##   <genre>Computer</genre>
##   <price>36.95</price>
##   <publish_date>2000-12-09</publish_date>
##   <description>Microsoft's .NET initiative is explored in
##         detail in this deep programmer's reference.</description>
## </book>
##
## [[7]]
## <book id="bk111" type="HardCover">
##   <author>O'Brien, Tim</author>
##   <title>MSXML3: A Comprehensive Guide</title>
##   <genre>Computer</genre>
##   <price>36.95</price>
##   <publish_date>2000-12-01</publish_date>
##   <description>The Microsoft MSXML3 parser is covered in
##         detail, with attention to XML DOM interfaces, XSLT processing,
##         SAX and more.</description>
## </book>
##
## attr(,"class")
## [1] "XMLNodeSet"
```

The *XML package* also offers functionality to convert the XML file to list or data frame format.

```
lib <- xmlToList(data)
lib
```

```
## $location
## [1] "HSSML"
##
## $catalog
## $catalog$book
## $catalog$book$author
## [1] "Gambardella, Matthew"
##
## $catalog$book$title
## [1] "XML Developer's Guide"
##
## $catalog$book$genre
## [1] "Computer"
##
## $catalog$book$price
## [1] "44.95"
##
## $catalog$book$publish_date
## [1] "2000-10-01"
##
## $catalog$book$description
## [1] "An in-depth look at creating applications \n        with XML."
##
## $catalog$book$.attrs
##          id        type
##     "bk101" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Ralls, Kim"
##
## $catalog$book$title
## [1] "Midnight Rain"
##
## $catalog$book$genre
## [1] "Fantasy"
##
## $catalog$book$price
## [1] "5.95"
##
## $catalog$book$publish_date
## [1] "2000-12-16"
##
## $catalog$book$description
## [1] "A former architect battles corporate zombies, \n        an evil sorceress, and her ow
n childhood to become queen \n        of the world."
##
## $catalog$book$.attrs
##          id        type
##     "bk102" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Corets, Eva"
##
```

```
## $catalog$book$title
## [1] "Maeve Ascendant"
##
## $catalog$book$genre
## [1] "Fantasy"
##
## $catalog$book$price
## [1] "5.95"
##
## $catalog$book$publish_date
## [1] "2000-11-17"
##
## $catalog$book$description
## [1] "After the collapse of a nanotechnology \n        society in England, the young surviv
ors lay the \n        foundation for a new society."
##
## $catalog$book$.attrs
##          id        type
##     "bk103" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Corets, Eva"
##
## $catalog$book$title
## [1] "Oberon's Legacy"
##
## $catalog$book$genre
## [1] "Fantasy"
##
## $catalog$book$price
## [1] "5.95"
##
## $catalog$book$publish_date
## [1] "2001-03-10"
##
## $catalog$book$description
## [1] "In post-apocalypse England, the mysterious \n        agent known only as Oberon helps
to create a new life \n        for the inhabitants of London. Sequel to Maeve \n        Ascen
dant."
##
## $catalog$book$.attrs
##          id        type
##     "bk104" "SoftCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Corets, Eva"
##
## $catalog$book$title
## [1] "The Sundered Grail"
##
## $catalog$book$genre
## [1] "Fantasy"
##
## $catalog$book$price
```

```
## [1] "5.95"
##
## $catalog$book$publish_date
## [1] "2001-09-10"
##
## $catalog$book$description
## [1] "The two daughters of Maeve, half-sisters, \n        battle one another for control of
England. Sequel to \n        Oberon's Legacy."
##
## $catalog$book$.attrs
##          id        type
##     "bk105" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Randall, Cynthia"
##
## $catalog$book$title
## [1] "Lover Birds"
##
## $catalog$book$genre
## [1] "Romance"
##
## $catalog$book$price
## [1] "4.95"
##
## $catalog$book$publish_date
## [1] "2000-09-02"
##
## $catalog$book$description
## [1] "When Carla meets Paul at an ornithology \n        conference, tempers fly as feathers
get ruffled."
##
## $catalog$book$.attrs
##          id        type
##     "bk106" "SoftCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Thurman, Paula"
##
## $catalog$book$title
## [1] "Splish Splash"
##
## $catalog$book$genre
## [1] "Romance"
##
## $catalog$book$price
## [1] "4.95"
##
## $catalog$book$publish_date
## [1] "2000-11-02"
##
## $catalog$book$description
## [1] "A deep sea diver finds true love twenty \n        thousand leagues beneath the sea."
##
```

```
## $catalog$book$.attrs
##          id        type
##     "bk107" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Knorr, Stefan"
##
## $catalog$book$title
## [1] "Creepy Crawlies"
##
## $catalog$book$genre
## [1] "Horror"
##
## $catalog$book$price
## [1] "4.95"
##
## $catalog$book$publish_date
## [1] "2000-12-06"
##
## $catalog$book$description
## [1] "An anthology of horror stories about roaches,\n        centipedes, scorpions  and oth
er insects."
##
## $catalog$book$.attrs
##          id        type
##     "bk108" "SoftCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Kress, Peter"
##
## $catalog$book$title
## [1] "Paradox Lost"
##
## $catalog$book$genre
## [1] "Science Fiction"
##
## $catalog$book$price
## [1] "6.95"
##
## $catalog$book$publish_date
## [1] "2000-11-02"
##
## $catalog$book$description
## [1] "After an inadvertant trip through a Heisenberg\n        Uncertainty Device, James Sal
way discovers the problems \n        of being quantum."
##
## $catalog$book$.attrs
##          id        type
##     "bk109" "SoftCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "O'Brien, Tim"
```

```
##
## $catalog$book$title
## [1] "Microsoft .NET: The Programming Bible"
##
## $catalog$book$genre
## [1] "Computer"
##
## $catalog$book$price
## [1] "36.95"
##
## $catalog$book$publish_date
## [1] "2000-12-09"
##
## $catalog$book$description
## [1] "Microsoft's .NET initiative is explored in \n        detail in this deep programmer's
reference."
##
## $catalog$book$.attrs
##          id        type
##     "bk110" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "O'Brien, Tim"
##
## $catalog$book$title
## [1] "MSXML3: A Comprehensive Guide"
##
## $catalog$book$genre
## [1] "Computer"
##
## $catalog$book$price
## [1] "36.95"
##
## $catalog$book$publish_date
## [1] "2000-12-01"
##
## $catalog$book$description
## [1] "The Microsoft MSXML3 parser is covered in \n        detail, with attention to XML DOM
interfaces, XSLT processing, \n        SAX and more."
##
## $catalog$book$.attrs
##          id        type
##     "bk111" "HardCover"
##
##
## $catalog$book
## $catalog$book$author
## [1] "Galos, Mike"
##
## $catalog$book$title
## [1] "Visual Studio 7: A Comprehensive Guide"
##
## $catalog$book$genre
## [1] "Computer"
##
## $catalog$book$price
```

```
## [1] "49.95"
##
## $catalog$book$publish_date
## [1] "2001-04-16"
##
## $catalog$book$description
## [1] "Microsoft Visual Studio 7 is explored in depth,\n        looking at how Visual Basic,
Visual C++, C#, and ASP+ are \n        integrated into a comprehensive development \n
environment."
##
## $catalog$book$.attrs
##          id        type
##     "bk112" "SoftCover"
```

```
Books <- xmlToDataFrame(books)
Books
```

```
##                 author                                    title    genre
## 1 Gambardella, Matthew                XML Developer's Guide Computer
## 2           Ralls, Kim                        Midnight Rain  Fantasy
## 3          Corets, Eva                     Maeve Ascendant  Fantasy
## 4          Corets, Eva                   The Sundered Grail  Fantasy
## 5       Thurman, Paula                        Splish Splash  Romance
## 6         O'Brien, Tim Microsoft .NET: The Programming Bible Computer
## 7         O'Brien, Tim         MSXML3: A Comprehensive Guide Computer
##   price publish_date
## 1 44.95   2000-10-01
## 2  5.95   2000-12-16
## 3  5.95   2000-11-17
## 4  5.95   2001-09-10
## 5  4.95   2000-11-02
## 6 36.95   2000-12-09
## 7 36.95   2000-12-01
##
description
## 1                                                                                An in-depth
look at creating applications \n        with XML.
## 2   A former architect battles corporate zombies, \n        an evil sorceress, and her own
childhood to become queen \n        of the world.
## 3   After the collapse of a nanotechnology \n        society in England, the young survivo
rs lay the \n        foundation for a new society.
## 4        The two daughters of Maeve, half-sisters, \n        battle one another for contro
l of England. Sequel to \n        Oberon's Legacy.
## 5                                                          A deep sea diver finds true love
twenty \n        thousand leagues beneath the sea.
## 6                                        Microsoft's .NET initiative is explored in \n
detail in this deep programmer's reference.
## 7 The Microsoft MSXML3 parser is covered in \n        detail, with attention to XML DOM in
terfaces, XSLT processing, \n        SAX and more.
```

We will be using the *rvest library* for webscraping.

Tidy Data

When data is tidy it means that each row represents one observation and the columns represent the different variables that we have data on for those observations.

Here is the code to import the data from the video.

```
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "fertility-two-countries-example.csv")
wide_data <- read_csv(filename)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   country = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
select(wide_data, country, '1960':'1967') #note it is in wide not tidy format
```

```
## # A tibble: 2 x 9
##   country     `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`
##   <chr>        <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Germany       2.41   2.44   2.47   2.49   2.49   2.48   2.44   2.37
## 2 South Korea   6.16   5.99   5.79   5.57   5.36   5.16   4.99   4.85
```

Wide formats allows for several observations in each row and also has one of the variables in the header - it is year in the example above.

Reshaping Data

We will make use of yet another tidyverse package, the *tidyr package*.

Wide to Tidy

Using the *gather()* function we can convert the above wide data into tidy data.

By default, the gather function will gather all the columns. Therefore, we will use the third argument in the function to specify the specific columns to gather. The first argument will be the name of column for the gathered variables - in this case, year. While the second argument is for the values in the column cells.

```
library(tidyr)
new_tidy_data <- wide_data %>% gather(year, fertility, '1960':'2015')
head(new_tidy_data)
```

```
## # A tibble: 6 x 3
##   country     year  fertility
##   <chr>       <chr>     <dbl>
## 1 Germany     1960       2.41
## 2 South Korea 1960       6.16
## 3 Germany     1961       2.44
## 4 South Korea 1961       5.99
## 5 Germany     1962       2.47
## 6 South Korea 1962       5.79
```

Note how the country column was not gather as it was not specified in the third argument of the *gather()* function.

Alternatively, we can specify the columns not to gather instead.

```
new_tidy_data <- wide_data %>% gather(year, fertility, -country)
head(new_tidy_data)
```

```
## # A tibble: 6 x 3
##   country     year  fertility
##   <chr>       <chr>     <dbl>
## 1 Germany     1960       2.41
## 2 South Korea 1960       6.16
## 3 Germany     1961       2.44
## 4 South Korea 1961       5.99
## 5 Germany     1962       2.47
## 6 South Korea 1962       5.79
```

```
class(new_tidy_data$year)
```

```
## [1] "character"
```

Also note that *gather()* function assumes the column names to be character atomic data type.

We can use R-bases *as.integer()* function to convert this or to use the *gather()* function's inbuilt convert keyword argument.

```
new_tidy_data <- wide_data %>% gather(year, fertility, -country, convert = TRUE)
head(new_tidy_data)
```

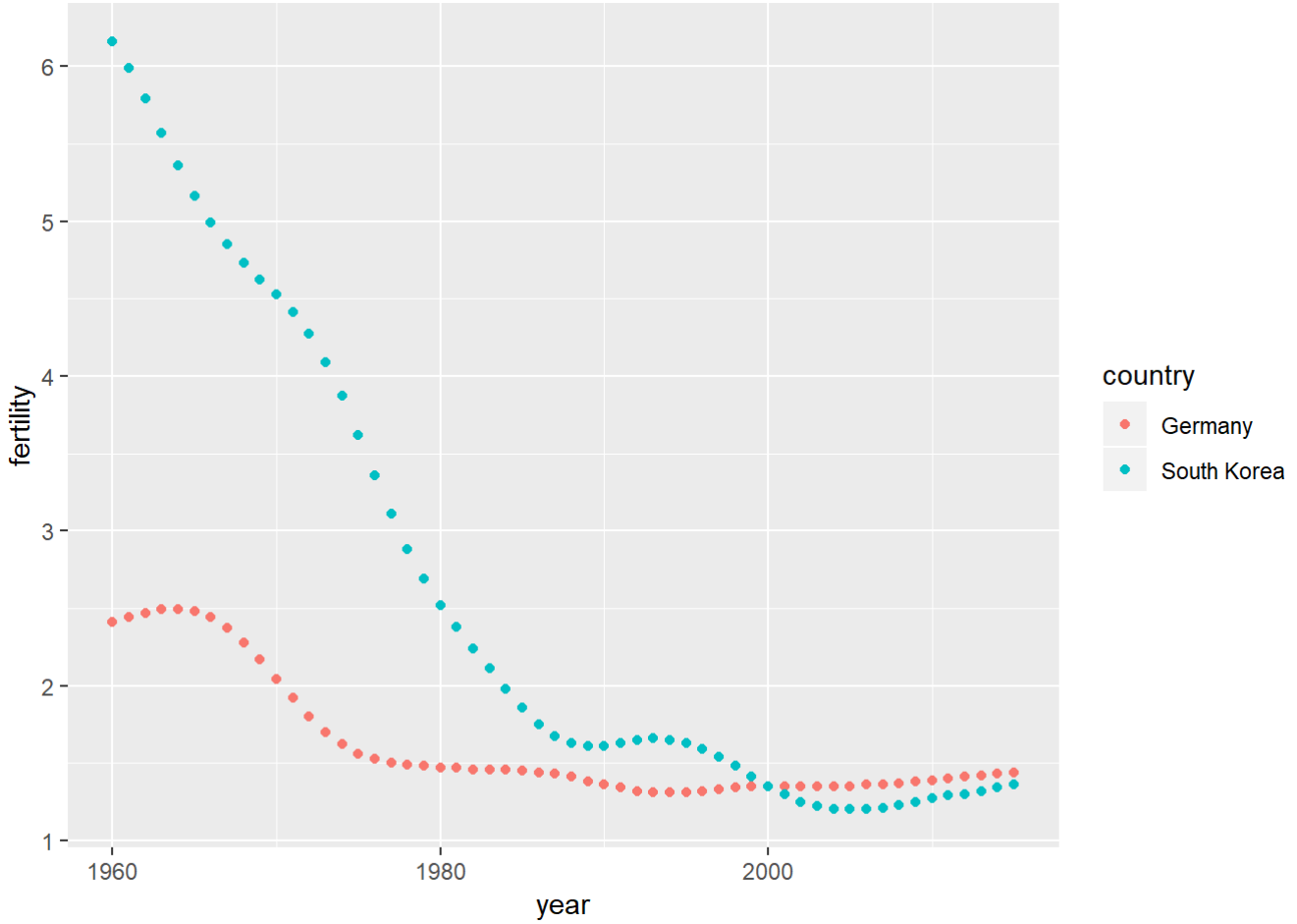
```
## # A tibble: 6 x 3
##   country      year fertility
##   <chr>       <int>     <dbl>
## 1 Germany      1960      2.41
## 2 South Korea  1960      6.16
## 3 Germany      1961      2.44
## 4 South Korea  1961      5.99
## 5 Germany      1962      2.47
## 6 South Korea  1962      5.79
```

```
class(new_tidy_data$year)
```

```
## [1] "integer"
```

Now we can plot prof Rafael's example plot with this tidy data!

```
library(ggplot2)
new_tidy_data %>% ggplot(aes(year, fertility, color = country)) + geom_point()
```



Tidy to Wide

Tidyr's spread() function does the inverse of the *gather()* function, converting tidy data to wide data. The first argument of the *spread()* function is to declare which variables are to be used as column names. While the second argument is to specify the variables used to fill out the cells.

```
new_wide_data <- new_tidy_data %>% spread(year, fertility)
select(new_wide_data, country, '1960':'1967')
```

```
## # A tibble: 2 x 9
##   country     `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`
##   <chr>        <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1 Germany       2.41   2.44   2.47   2.49   2.49   2.48   2.44   2.37
## 2 South Korea   6.16   5.99   5.79   5.57   5.36   5.16   4.99   4.85
```

Here is a cheat sheet (https://github.com/rstudio/cheatsheets/blob/master/data-import.pdf) on what we have learnt so far.

Separate and Unite

Usually, data is in a more complicated form. Below we have a file with more than one variable and stored widely and in a sub-optimal way.

```
path <- system.file("extdata", package = "dslabs")
filename <- file.path(path, "life-expectancy-and-fertility-two-countries-example.csv")
raw_dat <- read_csv(filename)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   country = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

```
select(raw_dat, 1:5)
```

```
## # A tibble: 2 x 5
##   country `1960_fertility` `1960_life_expe~ `1961_fertility`
##   <chr>              <dbl>            <dbl>            <dbl>
## 1 Germany             2.41             69.3             2.44
## 2 South ~             6.16             53.0             5.99
## # ... with 1 more variable: `1961_life_expectancy` <dbl>
```

Now we will use the gather function to convert our data from wide to tidy. However, notice that we can't name our column year as it also contains variable type. Instead, we will use the default key.

```
dat <- raw_dat %>% gather(key, value, -country)
head(dat)
```

```
## # A tibble: 6 x 3
##   country     key                  value
##   <chr>       <chr>                <dbl>
## 1 Germany     1960_fertility        2.41
## 2 South Korea 1960_fertility        6.16
## 3 Germany     1960_life_expectancy 69.3
## 4 South Korea 1960_life_expectancy 53.0
## 5 Germany     1961_fertility        2.44
## 6 South Korea 1961_fertility        5.99
```

Notice how much messier this is with more realistic data. It is not yet tidy as we should separate our fertility and life expectancy into separate columns.

```
dat$key[1:5]
```

```
## [1] "1960_fertility"       "1960_fertility"       "1960_life_expectancy"
## [4] "1960_life_expectancy" "1961_fertility"
```

The above shows how year and variable name are separated by underscore. Luckily, multiple variable names in a column are such a common problem that *readr* package has a suitable *separate()* function.

The *separate()* function requires the target column, the names for the new columns and the separator character. As the "_" underscore is the default separator character, we do not have to specify the third argument in this case.

Here we add an additional second variable name to catch the second half of "life_expectancy" due to the underscore separating them. The keyword argument fill lets us specify which column to place NA values when there isn't a third variable name.

```
dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), fill = "righ
t")
```

```
## # A tibble: 224 x 5
##    country     year  first_variable_name second_variable_name value
##    <chr>       <chr> <chr>               <chr>                <dbl>
##  1 Germany     1960  fertility           <NA>                  2.41
##  2 South Korea 1960  fertility           <NA>                  6.16
##  3 Germany     1960  life                expectancy           69.3
##  4 South Korea 1960  life                expectancy           53.0
##  5 Germany     1961  fertility           <NA>                  2.44
##  6 South Korea 1961  fertility           <NA>                  5.99
##  7 Germany     1961  life                expectancy           69.8
##  8 South Korea 1961  life                expectancy           53.8
##  9 Germany     1962  fertility           <NA>                  2.47
## 10 South Korea 1962  fertility           <NA>                  5.79
## # ... with 214 more rows
```

However, the keyword argument extra can also deal with this extra variable name from an unintended separate.

```
dat %>% separate(key, c("year", "variable_name"), extra = "merge")
```

```
## # A tibble: 224 x 4
##    country     year  variable_name   value
##    <chr>       <chr> <chr>           <dbl>
##  1 Germany     1960  fertility        2.41
##  2 South Korea 1960  fertility        6.16
##  3 Germany     1960  life_expectancy 69.3
##  4 South Korea 1960  life_expectancy 53.0
##  5 Germany     1961  fertility        2.44
##  6 South Korea 1961  fertility        5.99
##  7 Germany     1961  life_expectancy 69.8
##  8 South Korea 1961  life_expectancy 53.8
##  9 Germany     1962  fertility        2.47
## 10 South Korea 1962  fertility        5.79
## # ... with 214 more rows
```

Notice that the data is still not tidy, the variable_name column contains two types of variables and we still have the value column. We must use *spread()* on these remaining portions of the data frame.

```
dat %>% separate(key, c("year", "variable_name"), extra = "merge") %>% spread(variable_name,
 value)
```

```
## # A tibble: 112 x 4
##    country year  fertility life_expectancy
##    <chr>   <chr>     <dbl>           <dbl>
##  1 Germany 1960       2.41            69.3
##  2 Germany 1961       2.44            69.8
##  3 Germany 1962       2.47            70.0
##  4 Germany 1963       2.49            70.1
##  5 Germany 1964       2.49            70.7
##  6 Germany 1965       2.48            70.6
##  7 Germany 1966       2.44            70.8
##  8 Germany 1967       2.37            71.0
##  9 Germany 1968       2.28            70.6
## 10 Germany 1969       2.17            70.5
## # ... with 102 more rows
```

While not optimal in this case, we can use this example to showcase the *unite()* function.

```
dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), fill = "righ
t") %>% unite(variable_name, first_variable_name, second_variable_name) %>% spread(variable_n
ame, value) %>% rename(fertility = fertility_NA)
```

```
## # A tibble: 112 x 4
##    country year  fertility life_expectancy
##    <chr>   <chr>     <dbl>           <dbl>
##  1 Germany 1960       2.41            69.3
##  2 Germany 1961       2.44            69.8
##  3 Germany 1962       2.47            70.0
##  4 Germany 1963       2.49            70.1
##  5 Germany 1964       2.49            70.7
##  6 Germany 1965       2.48            70.6
##  7 Germany 1966       2.44            70.8
##  8 Germany 1967       2.37            71.0
##  9 Germany 1968       2.28            70.6
## 10 Germany 1969       2.17            70.5
## # ... with 102 more rows
```

Combining Data

Sometimes the information we need for an analysis might be in more than one table.

```
load("polls_us_election_2016.rda")
```

Here you can see that because the order of the states are not the same in both tables, we cannot simply join the two tables.

```
identical(results_us_election_2016$state, murders$state)
```

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

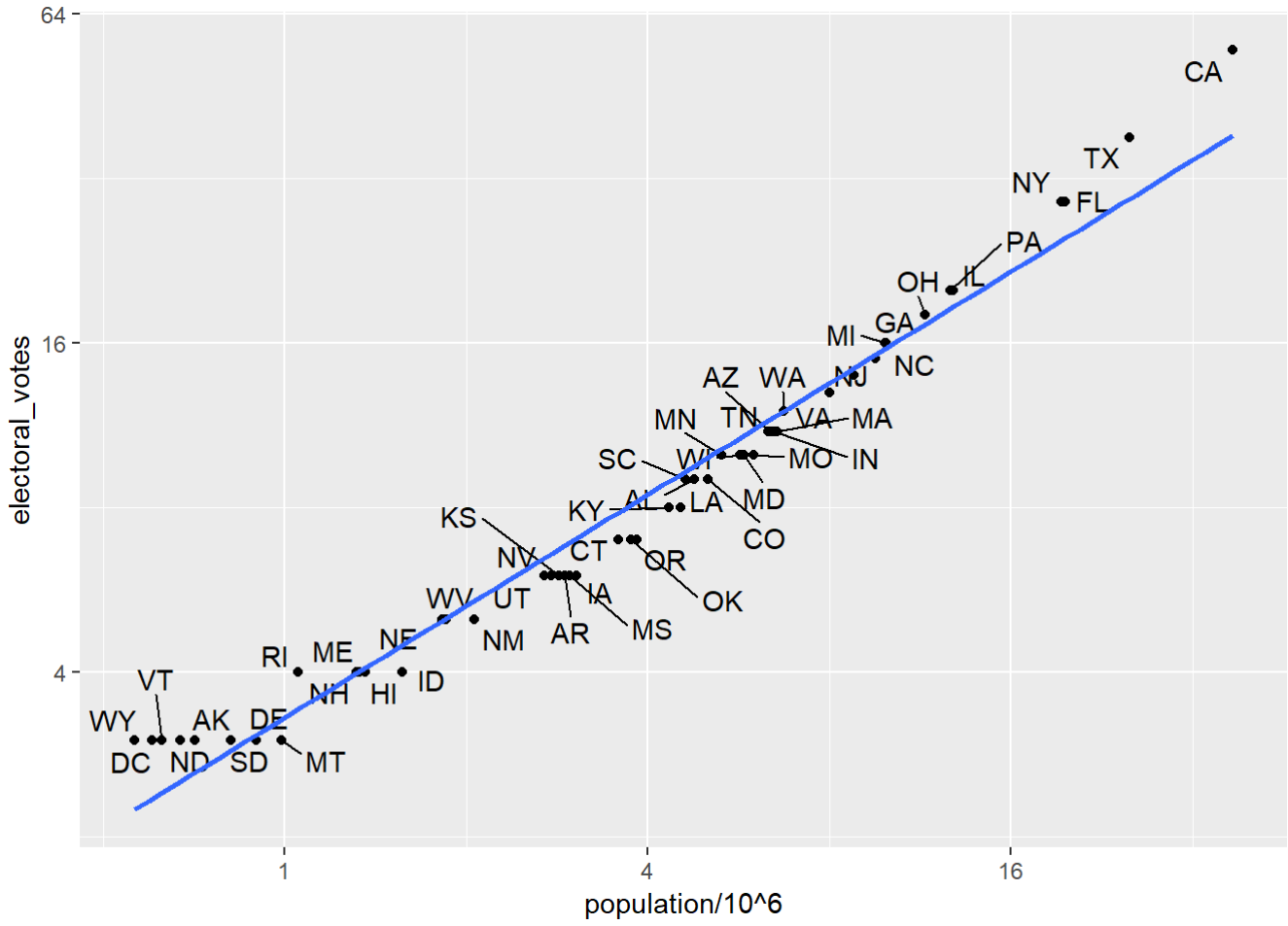
The *join()* functions in the *dplyr* package are based on the SQL joins (http://joins.spathon.com/) make sure that the tables are combined so that matching rows are together. We simply need to identify one or more columns that contain the information needed to match the two tables.

```
tab <- left_join(murders, results_us_election_2016, by = "state")
head(tab)
```

```
##        state abb region population total electoral_votes clinton trump
## 1    Alabama  AL  South    4779736   135               9    34.4  62.1
## 2     Alaska  AK   West     710231    19               3    36.6  51.3
## 3    Arizona  AZ   West    6392017   232              11    45.1  48.7
## 4   Arkansas  AR  South    2915918    93               6    33.7  60.6
## 5 California  CA   West   37253956  1257              55    61.7  31.6
## 6   Colorado  CO   West    5029196    65               9    48.2  43.3
##   others
## 1    3.6
## 2   12.2
## 3    6.2
## 4    5.8
## 5    6.7
## 6    8.6
```

Here is a plot of the combined data.

```
library(ggrepel)
tab %>% ggplot(aes(population/10^6, electoral_votes, label = abb)) +
  geom_point() +
  geom_text_repel() +
  scale_x_continuous(trans = "log2") +
  scale_y_continuous(trans = "log2") +
  geom_smooth(method = "lm", se = FALSE)
```



Usually we won't have a perfect match between two tables. We show this with subsets of both tables, so that the states in the subsets below will differ.

```
tab1 <- slice(murders, 1:6) %>% select(state,population)
tab1
```

```
##        state population
## 1    Alabama    4779736
## 2     Alaska     710231
## 3    Arizona    6392017
## 4   Arkansas    2915918
## 5 California   37253956
## 6   Colorado    5029196
```

```
tab2 <- slice(results_us_election_2016, c(1:3, 5, 7:8)) %>% select(state, electoral_votes)
tab2
```

```
##        state electoral_votes
## 1 California              55
## 2      Texas              38
## 3    Florida              29
## 4   Illinois              20
## 5       Ohio              18
## 6    Georgia              16
```

Left Join

Notice what happens when we attempt the *left_join()* function again.

```
left_join(tab1, tab2)
```

```
## Joining, by = "state"
```

```
##        state population electoral_votes
## 1    Alabama    4779736              NA
## 2     Alaska     710231              NA
## 3    Arizona    6392017              NA
## 4   Arkansas    2915918              NA
## 5 California   37253956              55
## 6   Colorado    5029196              NA
```

The *join()* functions can also take advantage of the %>% pipe operator.

```
tab1 %>% left_join(tab2)
```

```
## Joining, by = "state"
```

```
##        state population electoral_votes
## 1    Alabama    4779736              NA
## 2     Alaska     710231              NA
## 3    Arizona    6392017              NA
## 4   Arkansas    2915918              NA
## 5 California   37253956              55
## 6   Colorado    5029196              NA
```

Right Join

If we want the table like tab2, we can use the *right_join()* function

```
tab1 %>% right_join(tab2)
```

```
## Joining, by = "state"
```

```
##        state population electoral_votes
## 1 California   37253956              55
## 2      Texas         NA              38
## 3    Florida         NA              29
## 4   Illinois         NA              20
## 5       Ohio         NA              18
## 6    Georgia         NA              16
```

Inner Join

If we only want information from rows in both tables, that is like an intersection, we instead use the *inner_join()* function.

```
inner_join(tab1, tab2)   #notice that there will no NAs
```

```
## Joining, by = "state"
```

```
##        state population electoral_votes
## 1 California   37253956              55
```

Full Join

If we want to keep all rows in both tables, that is like a union, we instead use the *full_join()* function.

```
full_join(tab1, tab2)   #notice that there will be all the NAs
```

```
## Joining, by = "state"
```

```
##         state population electoral_votes
## 1     Alabama    4779736              NA
## 2      Alaska     710231              NA
## 3     Arizona    6392017              NA
## 4    Arkansas    2915918              NA
## 5  California   37253956              55
## 6    Colorado    5029196              NA
## 7       Texas         NA              38
## 8     Florida         NA              29
## 9    Illinois         NA              20
## 10       Ohio         NA              18
## 11    Georgia         NA              16
```

Semi Join

The *semi_join()* function allows us to keep the part of the first table for which we have information in the second table, but doesnt add the columns of the second.

```
semi_join(tab1, tab2)   #notice that there are less columns
```

```
## Joining, by = "state"
```

```
##        state population
## 1 California   37253956
```

Anti Join

The *anti_join()* function is the opposite of the *semi_join()* function. It allows us to keep the part of the first table for which we have **NO** information in the second table, but doesnt add the columns of the second.

```
anti_join(tab1, tab2)   #notice that there are less rows and columns
```

```
## Joining, by = "state"
```

```
##      state population
## 1  Alabama    4779736
## 2   Alaska     710231
## 3  Arizona    6392017
## 4 Arkansas    2915918
## 5 Colorado    5029196
```

Set Operators

These set operators work like union and intersect for vectors. If the *dplyr* package is loaded, these functions also work on the dataframe.

Intersect

We can take the *intersect()* of numeric or character vectors.

```
intersect(1:10, 6:15)
```

```
## [1]  6  7  8  9 10
```

```
intersect(c("a","b","c"), c("b","c","d"))
```

```
## [1] "b" "c"
```

However with *dplyr* package loaded, we can also do this for tables where they have the same column names.

```
library(dplyr)
tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
intersect(tab1, tab2)
```

```
##        state abb region population total electoral_votes clinton trump
## 1    Arizona  AZ   West    6392017   232              11    45.1  48.7
## 2   Arkansas  AR  South    2915918    93               6    33.7  60.6
## 3 California  CA   West   37253956  1257              55    61.7  31.6
##   others
## 1    6.2
## 2    5.8
## 3    6.7
```

Union

We can do the same with the *union()* function.

```
tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
union(tab1, tab2)
```

```
##         state abb    region population total electoral_votes clinton trump
## 1    Arkansas  AR     South    2915918    93               6    33.7  60.6
## 2     Arizona  AZ      West    6392017   232              11    45.1  48.7
## 3      Alaska  AK      West     710231    19               3    36.6  51.3
## 4  California  CA      West   37253956  1257              55    61.7  31.6
## 5     Alabama  AL     South    4779736   135               9    34.4  62.1
## 6    Colorado  CO      West    5029196    65               9    48.2  43.3
## 7 Connecticut  CT Northeast    3574097    97               7    54.6  40.9
##   others
## 1    5.8
## 2    6.2
## 3   12.2
## 4    6.7
## 5    3.6
## 6    8.6
## 7    4.5
```

Setdiff

We can also get the set difference with *setdiff()*. Unlike *intersect()* and *union()* function, the function is asymmetrical.

```
setdiff(1:10, 6:15)   #notice there are two different answers
```

```
## [1] 1 2 3 4 5
```

```
setdiff(6:15, 1:10)
```

```
## [1] 11 12 13 14 15
```

However with *dplyr* package loaded, we can also do this for tables where they have the same column names.

```
library(dplyr)
tab1 <- tab[1:5,]
tab2 <- tab[3:7,]
setdiff(tab1, tab2)
```

```
##     state abb region population total electoral_votes clinton trump others
## 1 Alabama  AL  South    4779736   135               9    34.4  62.1    3.6
## 2  Alaska  AK   West     710231    19               3    36.6  51.3   12.2
```

Setequal

The *setequal()* function helps us check if two sets are the same regardless of order.

```
setequal(1:5, 1:6)
```

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

```
setequal(1:5, 5:1)
```

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

However with *dplyr* package loaded, we can also do this for dataframes as well.

```
setequal(tab1, tab2)  #note the useful return output
```

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

Programming Structure and Functions

Dr. Liu Qizhang

5 February 2019

Introduction

This file covers the programming structures and functions in R that you may have encountered in other programming languages. Such as *Function()*, the R equivalent of python's def(). Iterating through your code with loops and program control flow with conditional statements are also covered.

Basic Conditionals

Below we have the most common conditional expression, the *if-else* statement. Notice that unlike python, we use curly braces "{}" to encapsulate our code as opposed to a colon ":" and whitespaces.

```
a <- 0
if(a!=0){
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
```

```
## [1] "No reciprocal for 0."
```

Note: While it is bad practice, notice that whitespaces do not affect our output, as R is a C-Family language.

```
a <- 0
if(a!=0){print(1/a)}else{print("No reciprocal for 0.")}
```

```
## [1] "No reciprocal for 0."
```

Notice how our output changes depending on what whether our a variable is true in the case of the if conditional or not. As it allows for us to control the flow of input to the desired expressions. In this case we do not want a divide by zero as it returns as positive infinity.

```
a <- 2
if(a!=0){
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
```

```
## [1] 0.5
```

```
print(1/0)
```

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

Below is the relevant pseudocode for clarity sake.

```
if(boolean condition){
  expressions
} else{
  alternative expressions
}
```

Code to load prof Rafael's dslab murders dataset. Remember to install the package first, if you haven't already with *install.packages("dslabs")*.

```
library(dslabs)
data(murders)
murder_rate<- murders$total/murders$population*100000
```

Below is an example demonstrating how we can use conditional statements to choose between displaying the stat with the lowest murder rate under a threshold, or to display that no state is under said threshold.

```
ind <- which.min(murder_rate)
if(murder_rate[ind] < 0.5){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
```

```
## [1] "Vermont"
```

```
ind <- which.min(murder_rate)
if(murder_rate[ind] < 0.25){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
```

```
## [1] "No state has murder rate that low"
```

A closely related function is the *ifelse()* function. To explain it better lets look at our pseudocode again.

```
if(boolean condition){
  expressions
} else{
  alternative expressions
}
```

Notice that the order is first a boolean condition, followed by the expression and then the alternative expression if the boolean condition returns as false. The *ifelse()* function follows this order without the use of curly braces.

```
ifelse(boolean condition,expressions,alternative expressions)
```

Here it is in action.

```
a <- 0
ifelse(a > 0, 1/a, NA)
```

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

It is especially useful because it also works on vectors, and returns the respective answers for each element.

```
a <- c(0,1,2,-4,5)
result <- ifelse(a > 0, 1/a, NA)
result
```

```
## [1]  NA 1.0 0.5  NA 0.2
```

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	0.25	NA	NA
5	TRUE	0.20	NA	0.2

Notice that we can't as easily parse the vector into the standard *if-else* conditional.

```
a <- c(0,1,2,-4,5)
if(a!=0){
  print(1/a)
} else{
  print(NA)
}
```

```
## Warning in if (a != 0) {: the condition has length > 1 and only the first
## element will be used
```

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

The dslabs na_example shows us how this *ifelse()* functions can be useful over the standard conditional when dealing with a dataset with NAs.

```
data(na_example)
sum(is.na(na_example))
```

```
## [1] 145
```

```
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
```

```
## [1] 0
```

Any Function

The *any()* function takes in a vector of logicals and returns TRUE if any of the logicals are true. Similar in concept to our logical OR "|" function.

```
z <- c(TRUE, TRUE, FALSE)
any(z)
```

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

```
z <- c(FALSE, FALSE, FALSE)
any(z)
```

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

All Function

The *all()* function takes in a vector of logicals and returns TRUE if all of the logicals are true. Similar in concept to our logical AND "&" function.

```
z <- c(TRUE, TRUE, FALSE)
all(z)
```

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

```
z <- c(TRUE, TRUE, TRUE)
all(z)
```

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

Basic Functions

Sometimes there are sets of computations that you will do frequently yet there isn't one neat predefined function in R-base or its packages. When that happens we can use the *function()* function to create our own, in python the equivalent is *def()*.

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

You can see in the above example that we are assigning it the name "avg" and that it only has one input x. Additionally, it computes the sum and length of the input and assignes it tos & n respectively. Finally it returns s divided by n.

Below you can see that our own function has an identical output to the predefined one.

```
x <- 1:100
avg(x)
```

```
## [1] 50.5
```

```
identical(mean(x), avg(x))
```

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

Variables defined inside a function are not saved in the workspace as their lexical scope is local only to the *avg()* function, meaning if we were to call s, we would get an object not found error.

```
s      #If this code chunk is run, i will get an error: object not found
```

Any assignments inside *function()* that we wrote ourselves in local in scope, meaning if we already had an s or n variable it wont get overwritten whenever we call *avg()*. The inverse is also true, we can define variables that have the same name as those inside a *function()* but they will not affect one another.

Notice how the *avg()* call does not return 0.3 and that when we call our s in the global scope it remains as 3.

```
s <- 3
avg(1:10)
```

```
## [1] 5.5
```

```
s
```

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

As before, here is the pseudocode for writing user defined functions.

```
my_function <- function(x){
  operations that operate on x which is defined by user of function
  value of final line is returned
}
```

Also, be aware that we are not just limited to a single input parameter.

```
my_function <- function(x, y, z){
  operations that operate on x, y, z which is defined by user of function
  value of final line is returned
}
```

As shown below, we can also set default arguments and in this case its default is TRUE.

```
avg <- function(x, arithmetic=TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
avg(1:10, FALSE)
```

```
## [1] 4.528729
```

For Loops

To check that the formula n*(n+1)/2 is really equivalent to the sum of all numbers from 1 to n, we can use the user defined functions we just previously learnt.

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
compute_s_n(3)
```

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

```
compute_s_n(100)
```

```
## [1] 5050
```

```
compute_s_n(2017)
```

```
## [1] 2035153
```

However, say we want to call our *compute_s_n()* function where n = 1:25, it does not make sense for us to write 25 lines calling the function where we only change n. Instead, for loops let us define the range our variable takes - in this case 1 to 25.

Here is the pseudocode.

```
for (i in range of values){
  operations that use i, which is changing across the range of values
}
```

Here it is in action.

```
for (i in 1:5){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Note that unlike our user defined functions before i is **NOT** in local scope, calling i afterwards gets you the last value in the range and that it may overwrite any variable you assign as i.

```
i
```

```
## [1] 5
```

```
i <- "important work"
for (i in 1:5){
  print(i)
}
```

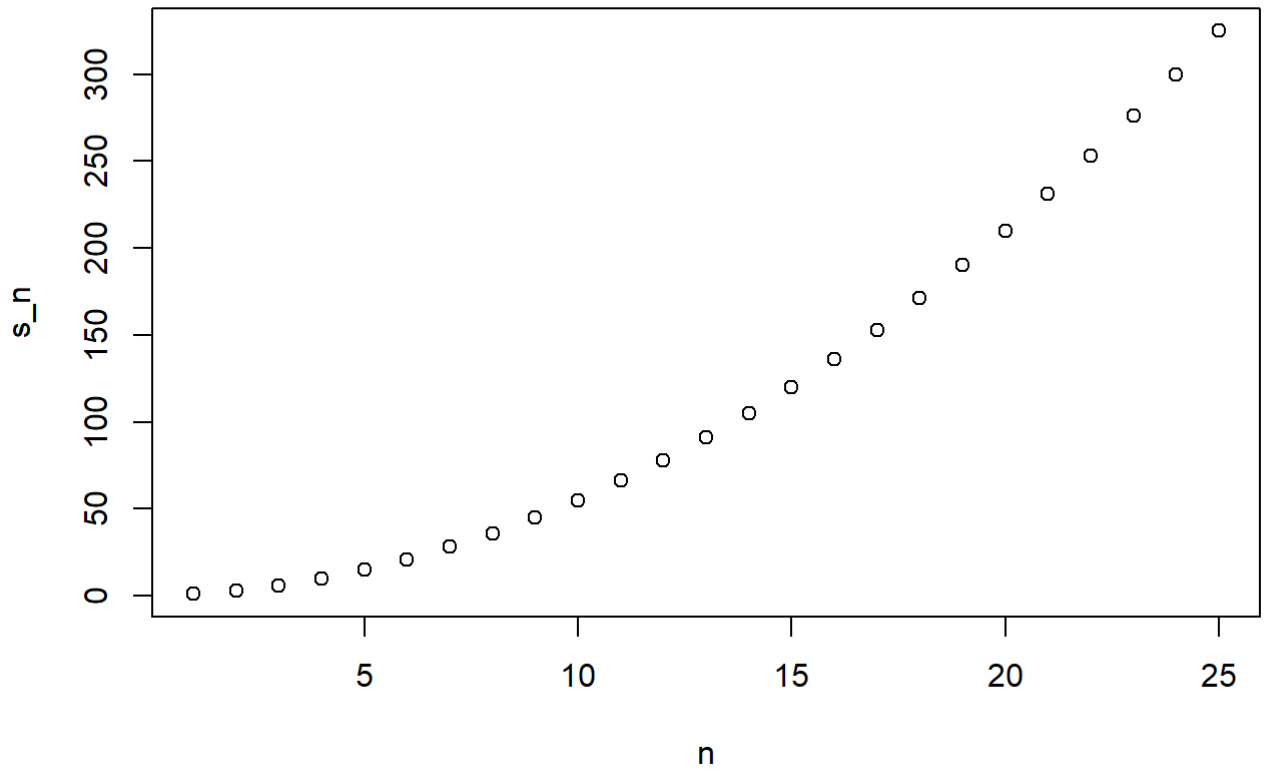
```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

```
i   #my important work is gone
```

```
## [1] 5
```

Back to our sums example, here is the code.

```
m <- 25
#create an empty vector
s_n <- vector(length = m)
for(n in 1:m){
  #store each result in its vector element with index of n
  s_n[n] <- compute_s_n(n)
}
n <- 1:m
plot(n, s_n)
```

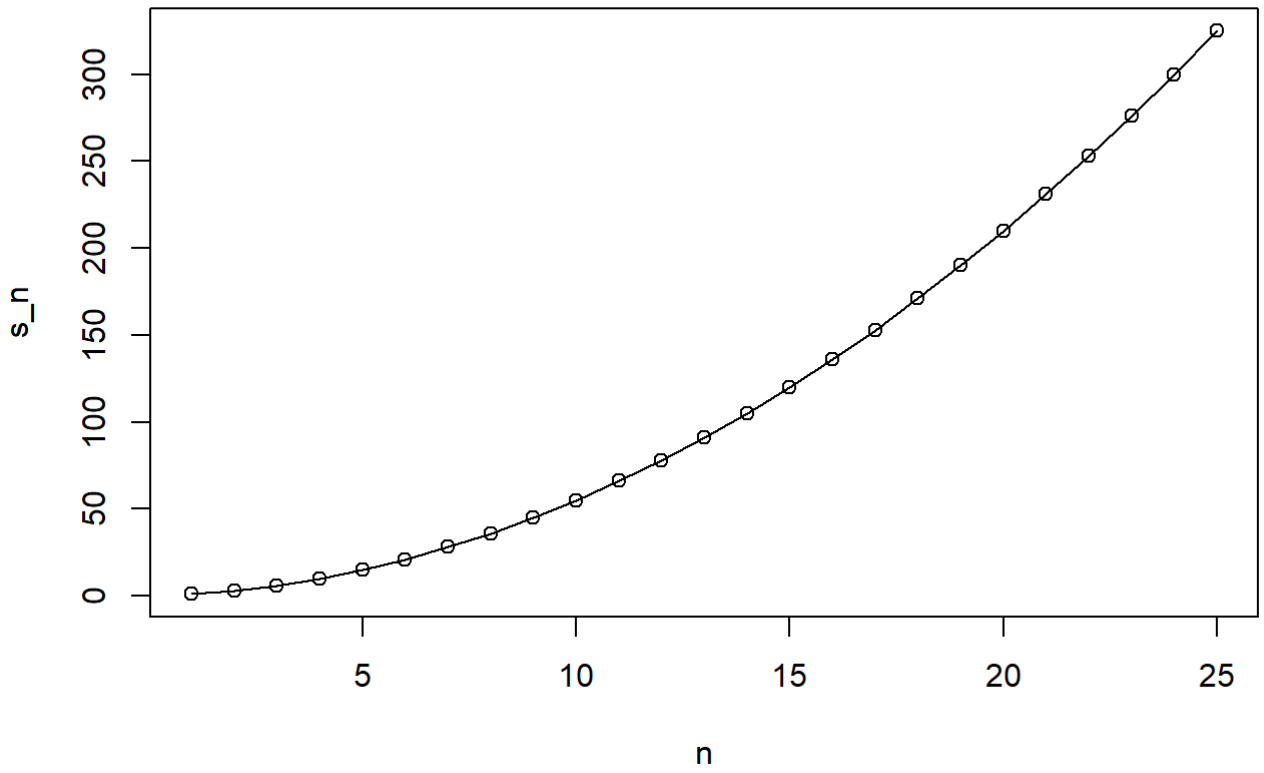


Comparing results from our for loop and the formula, you can see we get the same answers

n	s_n	n(n+1)/2
1	1	1
2	3	3
3	6	6
4	10	10
5	15	15
6	21	21
7	28	28
8	36	36
9	45	45
10	55	55

We can also show this graphically

```
plot(n, s_n)
lines(n, n*(n+1)/2)
```



Other Functions

While the concept of for loops are important, we rarely use for loops as there are more powerful predefined functions that we use instead. The apply family of functions - that is *apply()*, *lapply()*, *sapply()*, *vapply()*, *rapply()*, *mapply()*, *tapply()*.

Also, other useful functions are *split()*, *cut()*, *quantile()*, *reduce()*, *identical()* and *unique()*.

Apply Function

The *apply()* function allows you to apply a function to the margin of a matrix or a dataframe. The *apply()* function's parameters is as follows.

```
apply(x, MARGIN, FUNC, ...)
```

x is the target matrix/dataframe. MARGIN defines how the function is applied - whether it be across rows (=1) or columns (=2). FUNC is the function that you want applied to the data.

```
z <- cbind(A=1:3,B=4:6,C=7:9,D=10:12)
z
```

```
##      A B C  D
## [1,] 1 4 7 10
## [2,] 2 5 8 11
## [3,] 3 6 9 12
```

```
apply(z,2,sum)  #returns a vector of the column sums
```

```
##  A  B  C  D
##  6 15 24 33
```

Note that the FUNC in apply() must be able to accept a vector as its input variable.

User-defined functions will also work within apply family of functions.

```
CountOdd <- function(x) {
  sum(x%%2)
}
apply(z,2,CountOdd)
```

```
## A B C D
## 2 1 2 1
```

```
CountOddorEven <- function(x, flag) {
  ifelse(flag, sum(x%%2), length(x)-sum(x%%2))
}
apply(z,2,CountOddorEven, TRUE)
```

```
## A B C D
## 2 1 2 1
```

```
apply(z,2,CountOddorEven, FALSE)
```

```
## A B C D
## 1 2 1 2
```

Notice that when our FUNC function takes in more than one input parameter any additional parameters are specified after FUNC in the "…" position from earlier.

Lapply Function

The *lapply()* function works similarly to *apply()* except it works on list or vector inputs instead of matrix/dataframe input. Also, it always returns a list of the same length as the given list or array.

```
x <- list(A=1:4, B=seq(0.1,1,by=0.1))
x
```

```
## $A
## [1] 1 2 3 4
##
## $B
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
lapply(x, mean)
```

```
## $A
## [1] 2.5
##
## $B
## [1] 0.55
```

Sapply Function

It is a wrapper of the *lapply()* function. It also takes in a list or vector, however it returns a vector instead of a list

```
sapply(x, mean)
```

```
##    A    B
## 2.50 0.55
```

Vapply Function

The *vapply()* function performs exactly like *lapply()* except that we can specify the return value type from FUNC. This often means it is safer(harder to introduce NAs) and can be faster if we know that our output can use a atomic data type that takes up less memory space.

```
vapply(x, mean, numeric(1))
```

```
##    A    B
## 2.50 0.55
```

Rapply Function

The recursive apply, *rapply()* function applies a specified function to all elements of a list recursively.

```
x <- list(A=2,B=list(-1,3),C=list(-2,list(-5,6)))
x
```

```
## $A
## [1] 2
##
## $B
## $B[[1]]
## [1] -1
##
## $B[[2]]
## [1] 3
##
##
## $C
## $C[[1]]
## [1] -2
##
## $C[[2]]
## $C[[2]][[1]]
## [1] -5
##
## $C[[2]][[2]]
## [1] 6
```

```
rapply(x, function(x){x^2})  #returns a vector
```

```
##  A B1 B2 C1 C2 C3
##  4  1  9  4 25 36
```

Mapply Function

The *mapply()* function is the multivariate version of *lapply()*, meaning it can instead take multiple vectors as inputs. It then applies the specified FUNC to each element of the vectors.

```
mapply(rep, 1:5, c(4,4,4,4,4))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    1    2    3    4    5
## [3,]    1    2    3    4    5
## [4,]    1    2    3    4    5
```

```
matrix(c(rep(1,4), rep(2,4), rep(3,4), rep(4,4), rep(5,4)), nrow = 4, ncol = 5)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    1    2    3    4    5
## [3,]    1    2    3    4    5
## [4,]    1    2    3    4    5
```

Notice how much less work mapply is.

Tapply Function

The *tapply()* function is one of the most usefull in the apply family of functions. It applies the specified FUNC to each group of an array, grouped based on levels of certain factors.

```
x <- 1:10
y <- factor(c("A","A","A","B","B","B","B","C","C","C"))
tapply(x,y,sum)
```

```
##  A  B  C
##  6 22 27
```

To better show what is happening, here is a table.

Function	sum	sum	sum
x	1 2 3	4 5 6 7	8 9 10
y	A A A	B B B B	C C C

The function *sum()* is to different groups based on the different factors.

```
sum(1:3)
```

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

```
sum(4:7)
```

```
## [1] 22
```

```
sum(8:10)
```

```
## [1] 27
```

Pivot Table

We can easily analyse data in r by grouping data by different fields - this is similar to Pivot Table in Excel. More importantly, you can summarize the data with your own function for specific purposes, which can't be done in excel.

Lets load the murders dataset from the *dslabs* package again.

```
library(dslabs)
data(murders)
tapply(murders$total, murders$region, sum)
```

```
##     Northeast         South North Central          West
##          1469          4195          1828          1911
```

```
class(murders$region)
```

```
## [1] "factor"
```

Realize that this only works because regions are in the factors. This means that we cannot simply replace murders$region with murders$population as it is in numeric. However, with the code below we can bin our population into intervals - similar to how we binned things in excel. This essentially converts population into a vector of factors which now works with tapply.

```
class(murders$population)
```

```
## [1] "numeric"
```

```
cut(murders$population, breaks = c(0, 1e+06, 1e+07, 1e+08))
```

```
##  [1] (1e+06,1e+07] (0,1e+06]     (1e+06,1e+07] (1e+06,1e+07] (1e+07,1e+08]
##  [6] (1e+06,1e+07] (1e+06,1e+07] (0,1e+06]     (0,1e+06]     (1e+07,1e+08]
## [11] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+07,1e+08] (1e+06,1e+07]
## [16] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07]
## [21] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07]
## [26] (1e+06,1e+07] (0,1e+06]     (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07]
## [31] (1e+06,1e+07] (1e+06,1e+07] (1e+07,1e+08] (1e+06,1e+07] (0,1e+06]
## [36] (1e+07,1e+08] (1e+06,1e+07] (1e+06,1e+07] (1e+07,1e+08] (1e+06,1e+07]
## [41] (1e+06,1e+07] (0,1e+06]     (1e+06,1e+07] (1e+07,1e+08] (1e+06,1e+07]
## [46] (0,1e+06]     (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07] (1e+06,1e+07]
## [51] (0,1e+06]
## Levels: (0,1e+06] (1e+06,1e+07] (1e+07,1e+08]
```

```
tapply(murders$total, cut(murders$population, breaks = c(0, 1e+06, 1e+07, 1e+08)), mean)
```

```
##     (0,1e+06] (1e+06,1e+07] (1e+07,1e+08]
##       23.3750      134.3611      625.5714
```

Note: *tapply()* only works if the studied numbers are in **one vector**

Going back to the murders dataset, what if we want to find out the ratio of murder cases to population but separate this by region? It involves two different vectors murders$total and murders$population.

If we were to blindly apply this question to tapply…

```
tapply(murders$total/murders$population, murders$region, mean)
```

```
##     Northeast         South North Central          West
##  1.847500e-05  4.417012e-05  2.182040e-05  1.833401e-05
```

However, it gives us the average of the total to population ratio split by region, not the total to population ratio in each region.

Split Function

The *split()* function allows us to split a dataframe into a list of data frames based on a factor array. Below we are splitting the murders data set into dataframes of each of the four regions.

```
split(murders, murders$region)
```

```
## $Northeast
##            state abb    region population total
## 7    Connecticut  CT Northeast    3574097    97
## 20         Maine  ME Northeast    1328361    11
## 22 Massachusetts  MA Northeast    6547629   118
## 30 New Hampshire  NH Northeast    1316470     5
## 31    New Jersey  NJ Northeast    8791894   246
## 33      New York  NY Northeast   19378102   517
## 39  Pennsylvania  PA Northeast   12702379   457
## 40  Rhode Island  RI Northeast    1052567    16
## 46       Vermont  VT Northeast     625741     2
##
## $South
##                   state abb region population total
## 1                 Alabama  AL  South    4779736   135
## 4                Arkansas  AR  South    2915918    93
## 8                Delaware  DE  South     897934    38
## 9  District of Columbia  DC  South     601723    99
## 10                Florida  FL  South   19687653   669
## 11                Georgia  GA  South    9920000   376
## 18               Kentucky  KY  South    4339367   116
## 19              Louisiana  LA  South    4533372   351
## 21               Maryland  MD  South    5773552   293
## 25            Mississippi  MS  South    2967297   120
## 34         North Carolina  NC  South    9535483   286
## 37               Oklahoma  OK  South    3751351   111
## 41         South Carolina  SC  South    4625364   207
## 43              Tennessee  TN  South    6346105   219
## 44                  Texas  TX  South   25145561   805
## 47               Virginia  VA  South    8001024   250
## 49          West Virginia  WV  South    1852994    27
##
## $`North Central`
##            state abb        region population total
## 14      Illinois  IL North Central   12830632   364
## 15       Indiana  IN North Central    6483802   142
## 16          Iowa  IA North Central    3046355    21
## 17        Kansas  KS North Central    2853118    63
## 23      Michigan  MI North Central    9883640   413
## 24     Minnesota  MN North Central    5303925    53
## 26      Missouri  MO North Central    5988927   321
## 28      Nebraska  NE North Central    1826341    32
## 35 North Dakota  ND North Central     672591     4
## 36          Ohio  OH North Central   11536504   310
## 42 South Dakota  SD North Central     814180     8
## 50     Wisconsin  WI North Central    5686986    97
##
## $West
##          state abb region population total
## 2       Alaska  AK   West     710231    19
## 3      Arizona  AZ   West    6392017   232
## 5   California  CA   West   37253956  1257
## 6     Colorado  CO   West    5029196    65
## 12      Hawaii  HI   West    1360301     7
## 13       Idaho  ID   West    1567582    12
## 27     Montana  MT   West     989415    12
## 29      Nevada  NV   West    2700551    84
```

```
## 32 New Mexico  NM   West    2059179    67
## 38     Oregon  OR   West    3831074    36
## 45       Utah  UT   West    2763885    22
## 48 Washington  WA   West    6724540    93
## 51    Wyoming  WY   West     563626     5
```

We will also have to create a user-defined function to try to achieve our purpose.

```
murderRatio <- function(x)
{
  sum(x$total)/sum(x$population)
}
```

As we learnt earlier, we will have to use the *lapply()* function as it can take in lists or vectors instead of matrices or dataframes.

```
lapply(split(murders, murders$region), murderRatio)
```

```
## $Northeast
## [1] 2.655592e-05
##
## $South
## [1] 3.626558e-05
##
## $`North Central`
## [1] 2.731334e-05
##
## $West
## [1] 2.656175e-05
```

```
tapply(murders$total/murders$population, murders$region, mean)
```

```
##     Northeast         South North Central          West
##  1.847500e-05  4.417012e-05  2.182040e-05  1.833401e-05
```

Notice how the two results are different. We want our *tapply()* function to apply the region factors first, on both murders$total and murders$population and then divide one by the other, however in the naive first approach murders$total/murders$population is in fact done first before the tapply function is called. That is why we get such wildly different results, as the order in which we carry out our operations matter.

Group Data by Multiple Factors

We can easily group data by multiple factors using the *tapply()* function with the following.

```
tapply(mtcars$mpg, INDEX = list(mtcars$am, mtcars$gear), mean)
```

```
##          3      4     5
## 0 16.10667 21.050    NA
## 1       NA 26.275 21.38
```

Simulation

Dr. Liu Qizhang

1 September 2016

In this chapter, we are going to discuss how to use R to do Monte Carlo simulation, a mathematical technique for you to simulate all possible outcomes of variaous decisions subject to given probability distributions of certain decision factors, thus make optimal decision out of uncertainty. A framework to do simulation with R is given below:

Step 1. Modeling - Define a function to model one iteration of the simulation.

Step 2. Processing - run the simulation model for certain number of iterations.

Step 3. Summarizing - summarize the simulation results.

Step 4. Decision making - make your decision based on the simulation result.

Case 1. Birthday Problem

We will use the famous Birthday Problem to demonstrate how to build a simulationn model using R.

Version 1.

We start with the simplest version of the problem. Suppose that there are 30 people in a room. Ignoring the possibility of leap year, what is the probability that at least two of them have the same birthday?

```
#Define a function to run one iteration of the simulation.
iteration <- function(n){
    #We use 1 to 365 to represent the 365 days in a year. Now generate a series of 30 random
 number between 1 and 365 to represent the
    #birthdays of the 30 people
    birthdays <- sample(365, size=30,replace = TRUE)

    #Use rank function to rank the birthdays. If all birthdays are different, then rank will
 return 30 numbers from 1 to 30. Otherwise, it will return different
    #sequence of numbers. We use this property to tell if there is repeated birthdays.
    rank <- rank(birthdays, ties.method = "min")

    hasDuplication <- ifelse(sum(rank) == 30*(30+1)/2, 0, 1)

    return(hasDuplication)
}

#Now run the iteration for 1000 times and store the result in a vector
results <- sapply(1:1000, iteration)

#Calculate the probability
prob <- sum(results) / 1000

prob
```

```
## [1] 0.714
```

The above codes run the iteration 1000 times to get an estimated probability. However, we are not sure if this probability is close to the actual probability. We may use the following codes to achieve a probability accurate to 4 demical places.

```
#We create a function to calculate probability with number of iterations as input
simulate <- function(iters) {
    results <- sapply(1:iters,iteration)
    prob <- sum(results) / iters
    return (prob)
}

#Create a function to calculate the error of the simulation model with the given number of it
erations
SimError <- function(iters){
    prob1 <- simulate(iters)
    prob2 <- simulate(iters)

    return (abs(prob1-prob2))
}

GetDuplicateBirthDayProb <- function(error.threshold){
    #Now try to find out the desired probability
    num.iterations <- 1000

    while(num.iterations < 10000000) {
        error <- SimError(num.iterations)

        if(error < error.threshold)
            break

        num.iterations <- num.iterations *10
    }

    simulate(num.iterations)
}

GetDuplicateBirthDayProb(0.001)
```

```
## [1] 0.7063416
```

Version 2.

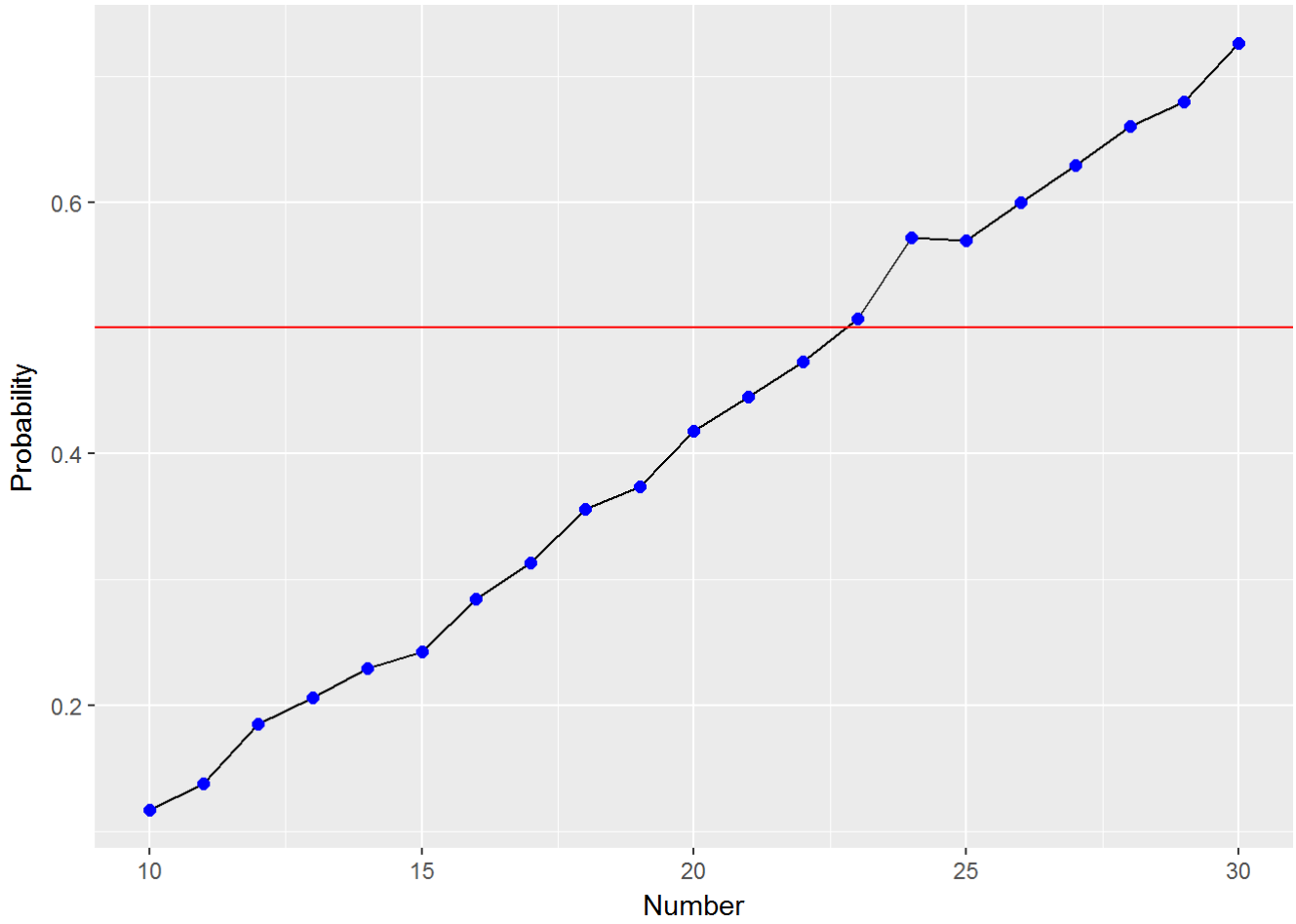
As you can tell from version 1 that the probability of having 2 duplicated birthdays among 30 people is more than 70%. We are now interested to know how this probability changes when the number of people changes. In particular, what is the minimum number of people needed to make the probability at least 50%?

In order to do this, we have to modify the functions above to make them more generic by making the number of people in the room as a parameter.

```
iteration <- function(n, numofpeople){
    #We use 1 to 365 to represent the 365 days in a year. Now generate a series of 30 random
 number between 1 and 365 to represent the
    #birthdays of the 30 people
    birthdays <- sample(365, size=numofpeople,replace = TRUE)

    #Use rank function to rank the birthdays. If all birthdays are different, then rank will
 return 30 numbers from 1 to 30. Otherwise, it will return different
    #sequence of numbers. We use this property to tell if there is repeated birthdays.
    rank <- rank(birthdays, ties.method = "min")

    hasDuplication <- ifelse(sum(rank) == numofpeople*(numofpeople+1)/2, 0, 1)

    return(hasDuplication)
}

#We create a function to calculate probability with number of iterations as input
simulate <- function(iters,numofpeople) {
    results <- sapply(1:iters,iteration,numofpeople)
    prob <- sum(results) / iters
    return (prob)
}

#Create a function to calculate the error of the simulation model with the given number of it
erations
SimError <- function(iters, numofpeople){
    prob1 <- simulate(iters,numofpeople)
    prob2 <- simulate(iters,numofpeople)

    return (abs(prob1-prob2))
}

GetDuplicateBirthDayProb <- function(numofpeople, threshold){
    #Now try to find out the desired probability
    num.iterations <- 1000

    while(num.iterations < 10000000) {
        error <- SimError(num.iterations,numofpeople)

        if(error < threshold)
            break

        num.iterations <- num.iterations *10
    }

    simulate(num.iterations,numofpeople)
}

probs<-sapply(10:30,GetDuplicateBirthDayProb,0.01)
df <- data.frame(Number = 10:30, Probability = probs)

library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 3.3.2
```

```
ggplot(df,aes(x=Number, y=Probability)) + geom_line() + geom_point(color='blue',size=2) + geo
m_abline(slope = 0, intercept = 0.5, color='red')
```



From the graph, we can tell that we just need 23 people to make the probability of having duplicated birthdays above 50%.

Version 3

In the previous two versions, we ignore the possiblity of leap year so that we can use 365 across the whole simulation model. Now let's try to make the model more realistic by taking the possibility of leap year into consideration. Can you try to enhance model 2 to achieve this?

Exploring Data

Dr. Liu Qizhang

12 February 2019

Introduction

This file covers data exploration, such as missing data, data exploration and a basic introduction to ggplot2. Below we have the code to import the two datasets used this week.

```
property <- read.csv("property.csv")
customers <- read.csv("customers.csv")
```

Missing Data

Sometimes we may simply choose to remove any observations where there are missing data, such as the following.

```
property <- property[complete.cases(property), ]
```

However, we may lose some important insight if we simply remove any incomplete observations. More importantly, blindly doing so without checking the data and even overwriting the original dataset can be dangerous.

```
dim(property)
```

```
## [1]  0 20
```

Notice that we now have zero observations and we will have to reimport the data because of our blind overwrite.

Causes of Missing Data

```
property <- read.csv("property.csv")
property[37:40, c("Property.Type.", "Developer.", "HDB.Town.")]
```

```
##    Property.Type.                            Developer.   HDB.Town.
## 37    Condominium Sim Lian JV (Punggol Central) Pte Ltd        <NA>
## 38    Condominium               Hong Leong Holdings Ltd        <NA>
## 39    HDB 5 Rooms                                   <NA>  Sembawang
## 40    HDB 5 Rooms                                   <NA>  Sembawang
```

The slice of data shows that logically each observation will have at least one field with an NA. It will either have NA for HDB Town if it is a condominium or an NA in the Developer field if it is an HDB.

Here we have **Structural Missing Data**, where data is designed to be missing for the specific nature of the business. Developers are not relevant to HDBs and HDB Towns are not relevant to condominiums.

It is important to understand the nature of missing data before we decide on a course of action to deal with them. We may be missing data due to **human error** in data entry or collection, **system error** in capturing data, or **loopholes** in the business process.

```
property[1:13, c("Built.Year.", "Bathrooms.", "Floor.")]
```

```
##    Built.Year. Bathrooms.    Floor.
## 1         2010          4    GROUND
## 2         2015          1    GROUND
## 3           NA          1      <NA>
## 4         2016          2      HIGH
## 5         2016          2      HIGH
## 6         2016          2       MID
## 7         2016         NA      <NA>
## 8         2016          2       MID
## 9         2016          2      <NA>
## 10        2016          1      <NA>
## 11        2016         NA      <NA>
## 12        2010         NA      <NA>
## 13        2014          2 PENTHOUSE
```

Notice how many of the fields have NA values yet these properties should have a built year, bathrooms or a floor. It is likely due to some loophole in the business process, allowing for human operators to omit crucial data that is not structurally missing.

Dealing with Missing Data

Before we can decide on how we deal with the missing data, it is usually a good idea to have an overview of the dataset. A good way is with the *summary()* function.

```
summary(customers)
```

```
##        id          gender  is.employed         income
##  Min.   :   2068   F:440   Mode :logical   Min.   : -7600
##  1st Qu.: 345667   M:560   FALSE:73        1st Qu.: 14200
##  Median : 693403           TRUE :599       Median : 34500
##  Mean   : 698500           NA's :328       Mean   : 53648
##  3rd Qu.:1044606                           3rd Qu.: 68900
##  Max.   :1414286                           Max.   :454500
##                                            NA's   :73
##      marital.status bought.product                        housing.type
##  Divorced   :155    Mode :logical   Homeowner free and clear    :157
##  Ever Single:233    FALSE:159       Homeowner with mortgage/loan:412
##  Married    :516    TRUE :841       Rented                     :375
##  Widowed    : 96                    NA's                       : 56
##
##
##
##  is.member         num.cards          age                district
##  Mode :logical   Min.   :0.000   Min.   :  0.0   WOODLANDS    : 29
##  FALSE:820       1st Qu.:1.000   1st Qu.: 38.0   BUKIT PANJANG: 28
##  TRUE :124       Median :2.000   Median : 50.0   BOON LAY     : 26
##  NA's :56        Mean   :1.916   Mean   : 51.7   MARINA EAST  : 25
##                  3rd Qu.:2.000   3rd Qu.: 64.0   OUTRAM       : 24
##                  Max.   :6.000   Max.   :146.7   SENGKANG     : 24
##                  NA's   :56                      (Other)      :844
##  no.family.members           industry
##  Min.   :1.000     IT            :104
##  1st Qu.:2.000     Finance       :100
##  Median :4.000     Others        : 99
##  Mean   :4.485     Transportation: 97
##  3rd Qu.:7.000     Education     : 95
##  Max.   :8.000     (Other)       :177
##                    NA's          :328
```

The *summary()* function helpfully tells us how much data is missing in each field with its “NA’s” row.

We may have **missing categorical data**, such as in the case of the “is.employed” field. An easy solution would be to set a new category called “Unknown” for these missing records.

```
attach(customers)
customers$employment.status <- ifelse(is.na(is.employed),
                                      "Unknown",
                                      ifelse(is.employed,
                                             "Employed",
                                             "Unemployed"))
```

Notice that we choose to create a new field called “employment.status” instead of adding “Unknown” into the existing field “is.employed”. This is so that we retain our original data and can easily investigate should anything go wrong.

If we have **missing numerical data**, then it will be a bit more complex. Should data be missing randomly, then we may assign the mean value of the existing data to it or we can assign a value based on its relationship to other related variables.

We may be able to estimate missing income data with age or profession.

Sometimes, we may even have numerical missing data might be **structural**. For example, those below 20 or above 65 might not have any income at all.

Depending on the research objective, we can assign zeros to the missing data so that we can analyze our data numerically. Since these age groups will not be working, their income will indeed be 0.

```
customers$income.fix <- ifelse(is.na(income), 0, income)
```

However, if our interest is in different income levels, we might convert the numerical data into categorical data.

We will first create a vector representing the thresholds to classify income groups. This is like specifying our bin ranges in excel.

```
b <- c(0, 10000, 50000, 100000, 250000, 1000000)
```

Next, we will use the *cut()* function to break incomes into income groups and then convert the result into the character data type.

```
income.groups <- cut(income, breaks=b, include.lowest = TRUE)
income.groups <- as.character(income.groups)
```

Now, we will then name the missing data as “No Income”.

```
income.groups <- ifelse(is.na(income.groups),
                        "No Income",
                        income.groups)
```

Finally, we assign this vector as a new variable in the customers data frame.

```
customers$income.groups <- income.groups
```

These are just some common methods to handle missing data, the method to use depends on the nature of the data, your research interest, and your understanding of the data.

As you try this out yourself, remember to “detach” the customers dataset afterward.

```
detach(customers)
```

Other Data Problems

While missing data is the most common data problem we may face, it is considered an explicit data problem and as such, relatively easy to identify and deal with.

There can be other data problems that are hard to identify and yet may have a deep impact on our research if we do not handle it properly.

Data problems are normally caused by data entry mistakes, logical error, outdated data or inconsistent standards.

Data entry problems are usually easily spotted, notice that the minimum income below is -7600. It is very likely a data entry mistake, however, it is possible that debt may be recorded as negative income in this dataset. This needs to be clarified with the data provider, if possible.

Logical errors in data could be due to logical mistakes in the system the data was generated from. It could also be mishandling by a data processor if it is secondary data.

Strong domain knowledge and logical thinking are necessary to identify such a data problem.

Outdated data, even if perfect in quality, cannot be used in context-based and time-sensitive projects. An example is in predicting stock market movement.

Inconsistent data can result from multiple standards co-existing in a single dataset. This may occur as data originates from different sources, or have different people processing it.

Inconsistent data resulted in NASA losing its $125 million Mars Climate Orbiter in 1999.

A navigation team at the Jet Propulsion Laboratory used the Metric System's millimeters and meters in its calculations. Lockheed Martin Astronautics in Denver, who designed and built the spacecraft, provided crucial acceleration data in the Imperial System's inches, feet, and pounds.

As a result, JPL engineers mistook acceleration readings measured in Imperial units of pound-seconds for a metric measure of force called newton-seconds.

In a sense, the spacecraft was lost because of data inconsistency.

Data Visualization

Being able to communicate in data sense is an essential skill. Stephen Few, a famous data visualization expert, pointed out: "Numbers have an important story to tell. They rely on you to give them a clear and convincing voice." Findings from data projects can only be convincing and persuasive if presented properly. Visualization is the most effective way to do so. In Scott Berinato's HBR article (https://hbr.org/2016/06/visualizations-that-really-work), he classifies data visualization into four main types: idea illustration, idea generation, everyday DataViz, and visual discovery.

Visual discovery is probably the most valuable type of data visualization. Its goals include trend spotting, sense-making, and deep analysis.

In descriptive analytics, mentions of data visualizations usually refer to visual discovery. A good example of the power of visual discovery was in the 2016 rogue train case (https://blog.data.gov.sg/how-we-caught-the-circle-line-rogue-train-with-data-79405c86ab6a)

Principles of Data Visualization

There are eight core principles, as specified by Stephen Few.

Principle 1 - **Simplify**, it is not the more the better. Good data visualization capture the essence of data.

Principle 2 - **Compare**, it is always good to compare data visualizations side by side. The audience will be able to spot trends, patterns or differences with their eyes.

Principle 3 - **Attend**, good data visualization should highlight details we need the audience to attend to.

Principle 4 - **Explore**, good data visualization should allow the audience to explore data and discover things visually.

Principle 5 - **View diversely**, it is helpful to look at the same data from different perspectives concurrently and see how they fit together.

Principle 6 - **Ask why**, good data visualization should allow audiences to dig into the facts and find out why things are happening.

Principle 7 - **Be skeptical**, don't be content with the first answer we get. Drill down into the data and explore further.

Principle 8 - **Respond**, just exploring the data alone has limited benefit. Being able to share your data visually will lead to global enlightenment.

Ggplot2 Package

We will be making use of the ggplot2 package from the tidyverse family of packages. We can either install and load *tidyverse* or *ggplot2*. Last week, we already made use of the plotting capabilities that come in-built with R.

However, ggplot2 allows us to make complex and aesthetically pleasing plots quickly and intuitively from a beginner's standpoint. This is possible with the 'grammar of graphics' - the gg in ggplot. With a relatively modest set of ggplot building blocks and grammar, we can create hundreds of plots. Its default plot behavior is also designed to satisfy a majority of cases whilst still aesthetically pleasing.

One limitation is that ggplot is build to work exclusively with data tables - where rows have to be observations and columns variables. We will have to memorize the names of several functions and arguments, as such a ggplot2 cheatsheat (https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf) is recommended.

Graph Components

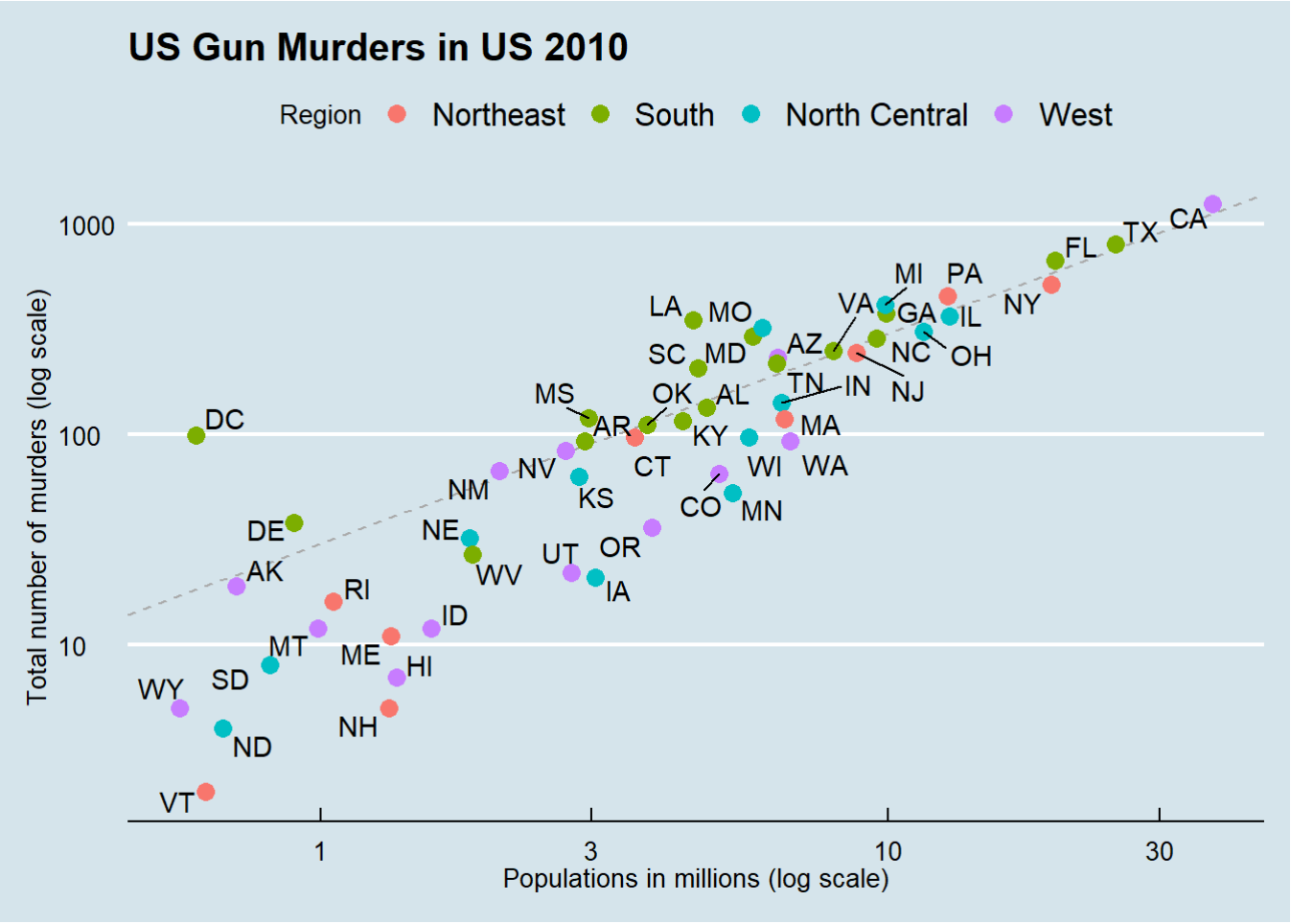
We will make use of dslab's murders dataset to look at graph components.

```
library(dslabs)
data(murders)
```

Also ensure that either *tidyverse* or *ggplot2* is installed and loaded, in my case its ggplot2.

```
library(ggplot2)
```

As motivation, below is the target graph we want to make.



The first step in learning ggplot2 is to be able to break a graph into components. The main *three* components we have to be aware of are:

1. The **data component**, the US murders data table in the example plot is summarized.
2. The **geometry component**, the example plot is a scatter plot. We also have bar plots histograms, smooth densities, q-q plots, and blocks plots.
3. The **aesthetic mapping component**, the x-axis values are used to display population size, while the y-axis values are used to display the total number of murders. Text is used to identify the states, and colors are used to denote the four different regions. These mappings depend on what geometry is used

Other components worth mentioning:

4. The **scale component**, the range of x-axis and y-axis appear to be defined by the range of the data, and in this case, are both on log scales.
5. The **labels, title, legend, etc**, the example plot uses the style used by The Economist magazine.

Creating a New Plot

Firstly, in order to create a ggplot graph, we must define a ggplot object. This is done with the *ggplot()* function, which initializes the graph.

The first argument in the *ggplot()* function is used to specify the graph's data component.

```
ggplot(data = murders)
```


This associates the dataset with the plotting object. However, we can also pipe the data equivalently.

```
murders %>% ggplot()
```

These both render a plot, however they are blank with a grey style background due to the lacking geometry component. As they were not assigned, they are automatically evaluated.

```
p <- ggplot(data = murders)
class(p)
```

```
## [1] "gg"     "ggplot"
```

When assigned instead, we must now either use the *print()* function or simply call the variable.

```
print(p)
```

p

Layers

In ggplot, we create graphs by adding layers. This is done as a component by component. These layers can define geometries, compute summary statistics, define what scales to use, and even change styles. To add layers, we use the symbol plus "+". Below we have some typical pseudocode.

```
DATA %>% ggplot() + LAYER 1 + LAYER 2 + ... + LAYER N
```

Usually, layer 1 defines the geometry component. As we want to make a scatterplot in the example plot, we will be using ggplot's *geom_point()* function. Geometry functions, in general, follow the pattern of "geom_" + name reminiscent to the geometry.

The *geom_point()* function, needs both the data component and the aesthetic mapping component. We have already connected the data component - the murders data table. To understand what mappings are required, we read the aesthetics section of the help file.

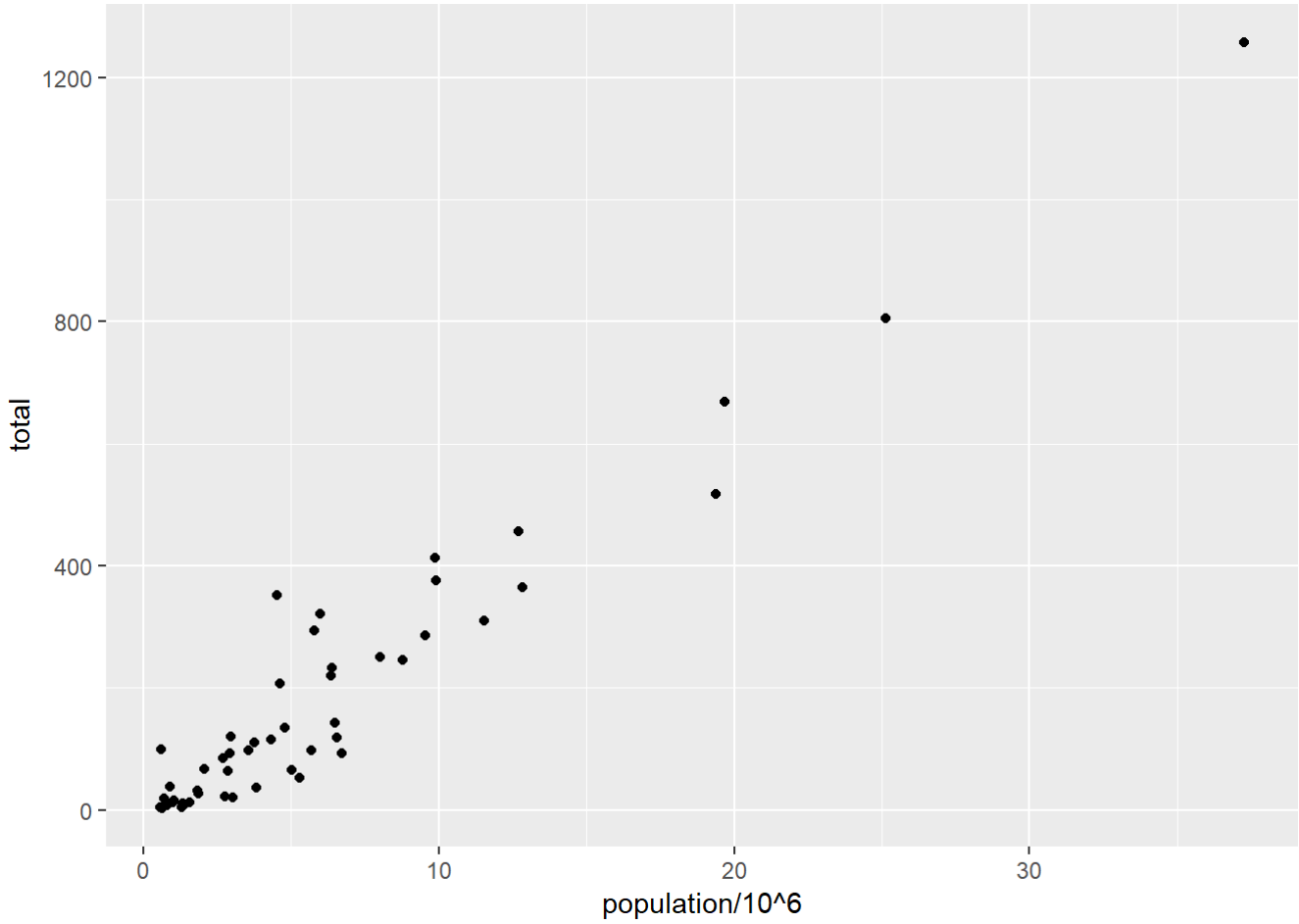
```
?geom_point
```

The required aesthetics under the aesthetics section are in bold. In this case, they are x and y - the x and y of the plot.

The *aes()* function connects data with what we see on the graph - this connection is called the aesthetic mappings. The output of this function is usually used as an argument of a geometry function.

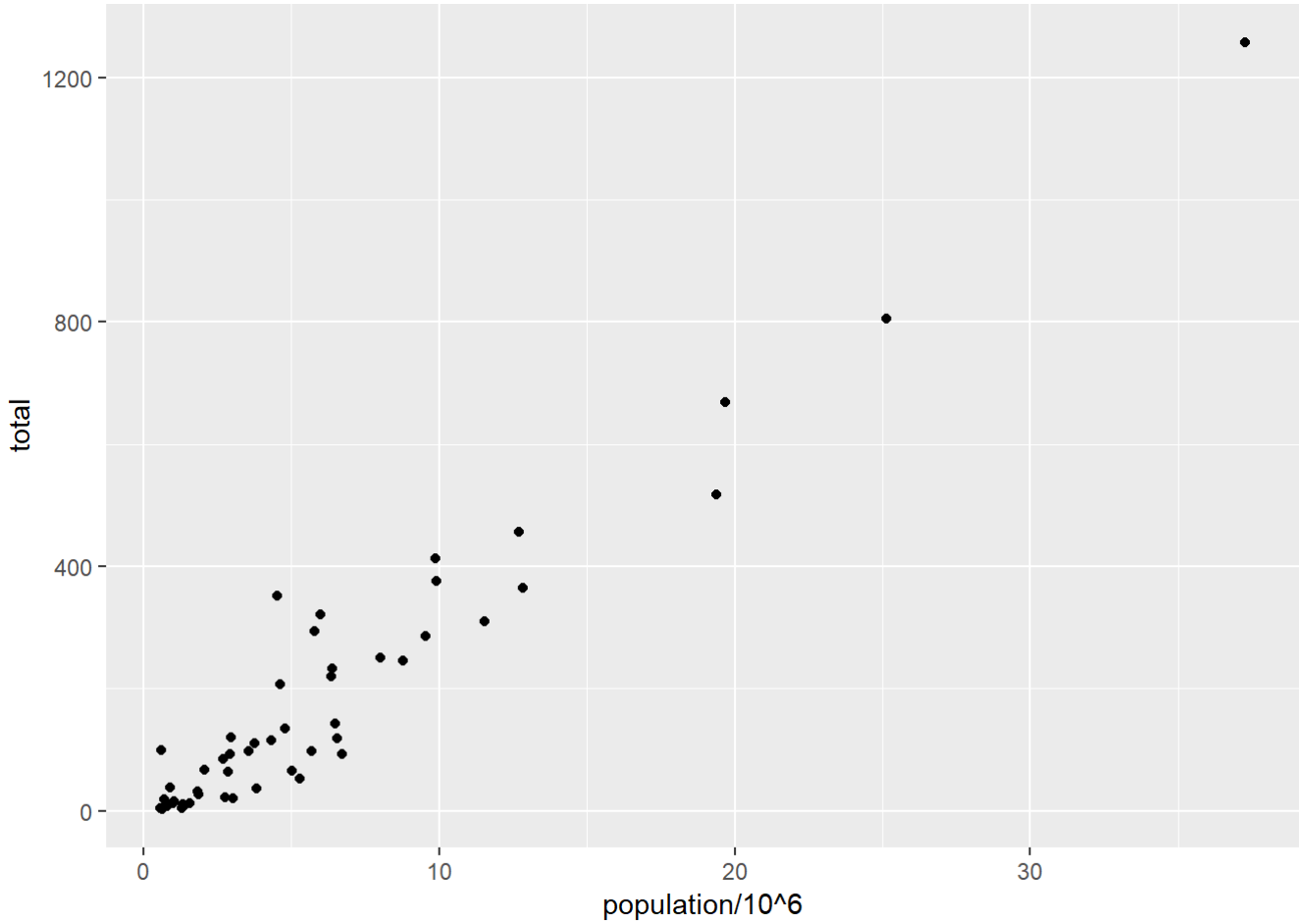
The following example produces a scatterplot of total murders versus population in millions.

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



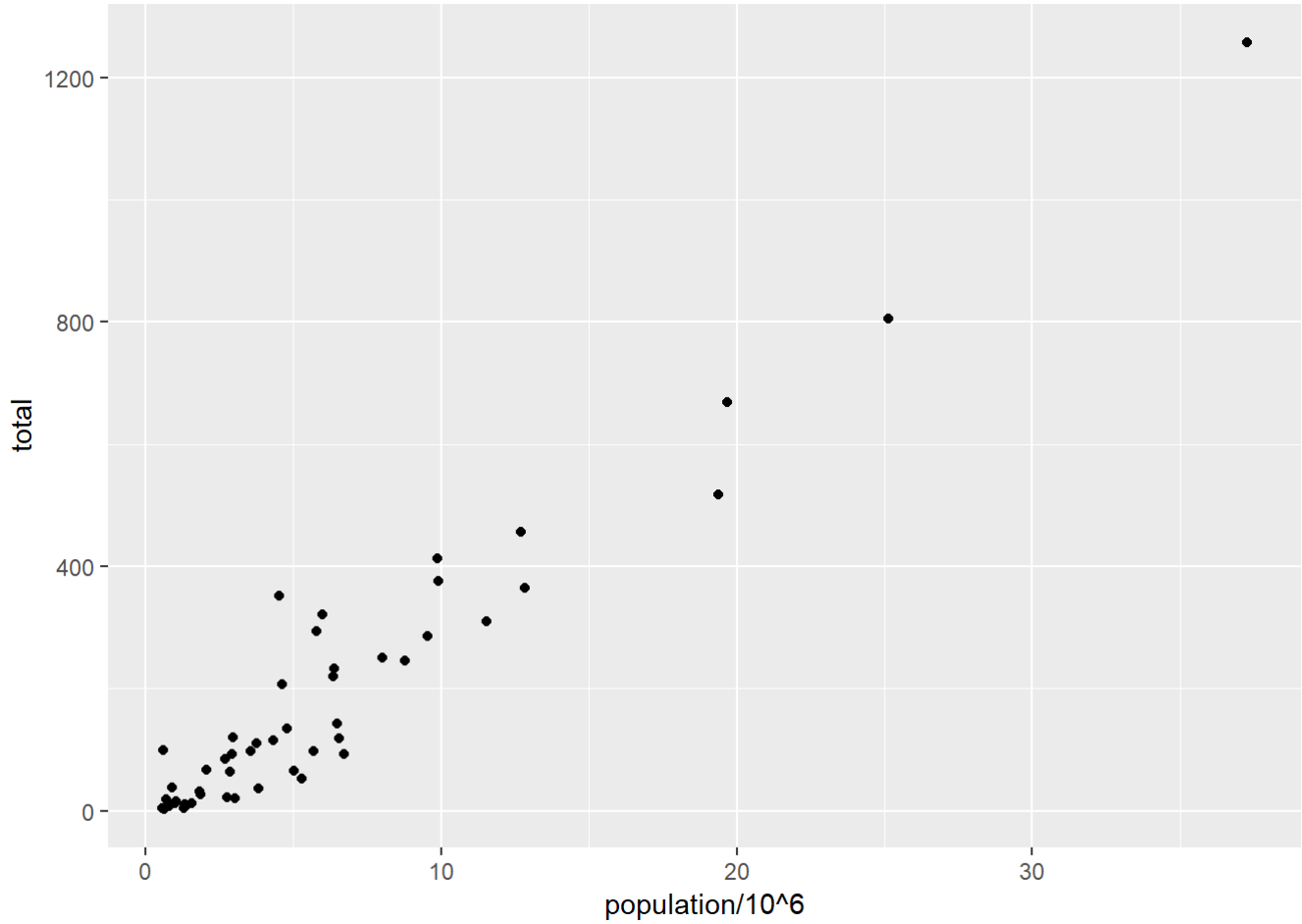
We pipe the murders dataset into ggplot, and then we add a layer with a *geom_point()* function. We can see that we're assigning population to x and total to y. We can actually drop the 'x' and the 'y' as they are the first and second expected arguments.

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



In ggplot, we can also add layers to previously defined objects.

```
p <- ggplot(data = murders)
p + geom_point(aes(population/10^6, total))
```



Note that this will render the object, as we did not assign it to a new object in the last line above.

The scales and labels are defined by default when adding this layer - the variable names from the object component are used to label the axes. The *aes()* function recognizes variables from the data component. This is specific to *aes()*.

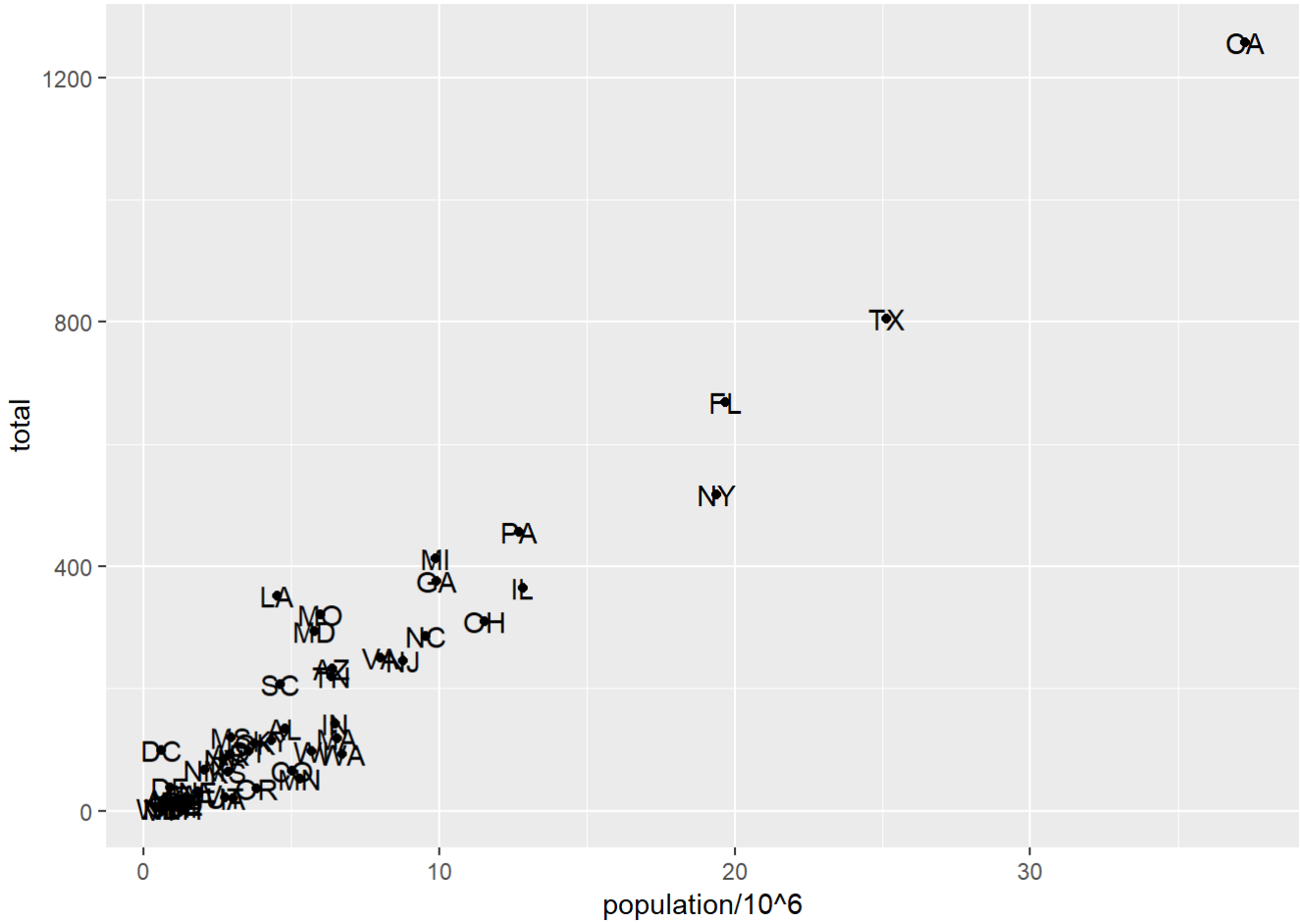
The second layer we wish to make involves adding a label to each point, as it helps us identify the state the point denotes. We will use the *geom_label()* and *geom_text()* functions to add text to the plot. Use label for a label with a rectangle and text to simply add text.

As each point needs a label, we will have to use the aesthetic mapping component to make this connection. Again, let's look at the help file.

```
?geom_text
```

The aesthetics section tells us that we supply the mapping between point and label through the label argument of aes.

```
p + geom_point(aes(population/10^6, total)) +
  geom_text(aes(population/10^6, total, label = abb))
```



We use the state abbreviations, stored in the “abb” variable.

Note that if we were to move label out of aes, we will get an error, as “abb” will not be found out of aes - not a globally defined variable.

```
p_test <- p + geom_text(aes(population/10^6, total), label = abb)
```

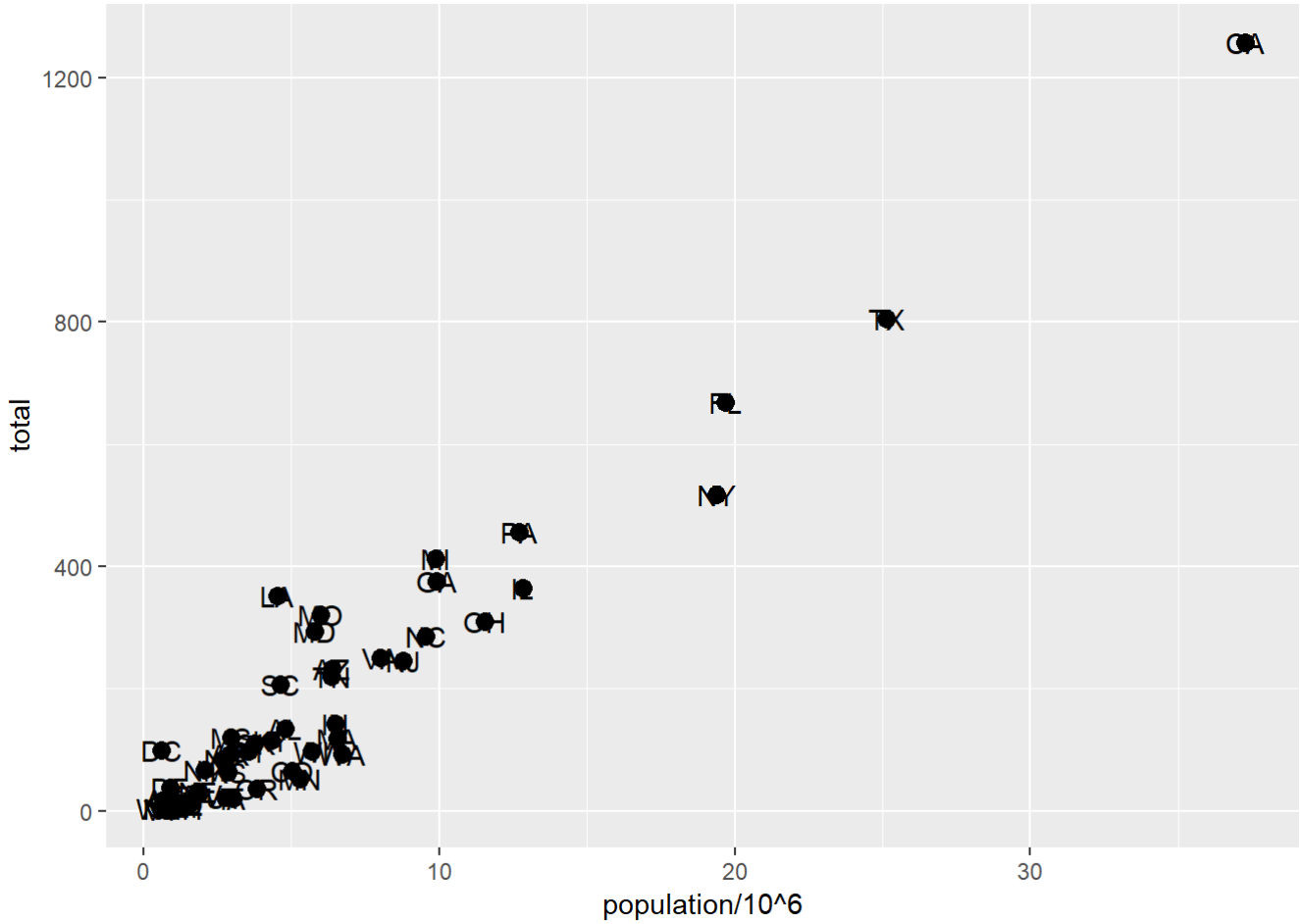
Tinkering

Each geometry function in ggplot has many other arguments other than aes and data, however, they tend to be specific to the plot. We wish to make the points larger than the default.

```
?geom_point
```

We can see that size is an aesthetic and can be changed using the code below.

```
p + geom_point(aes(population/10^6, total), size = 3) +
  geom_text(aes(population/10^6, total, label = abb))
```

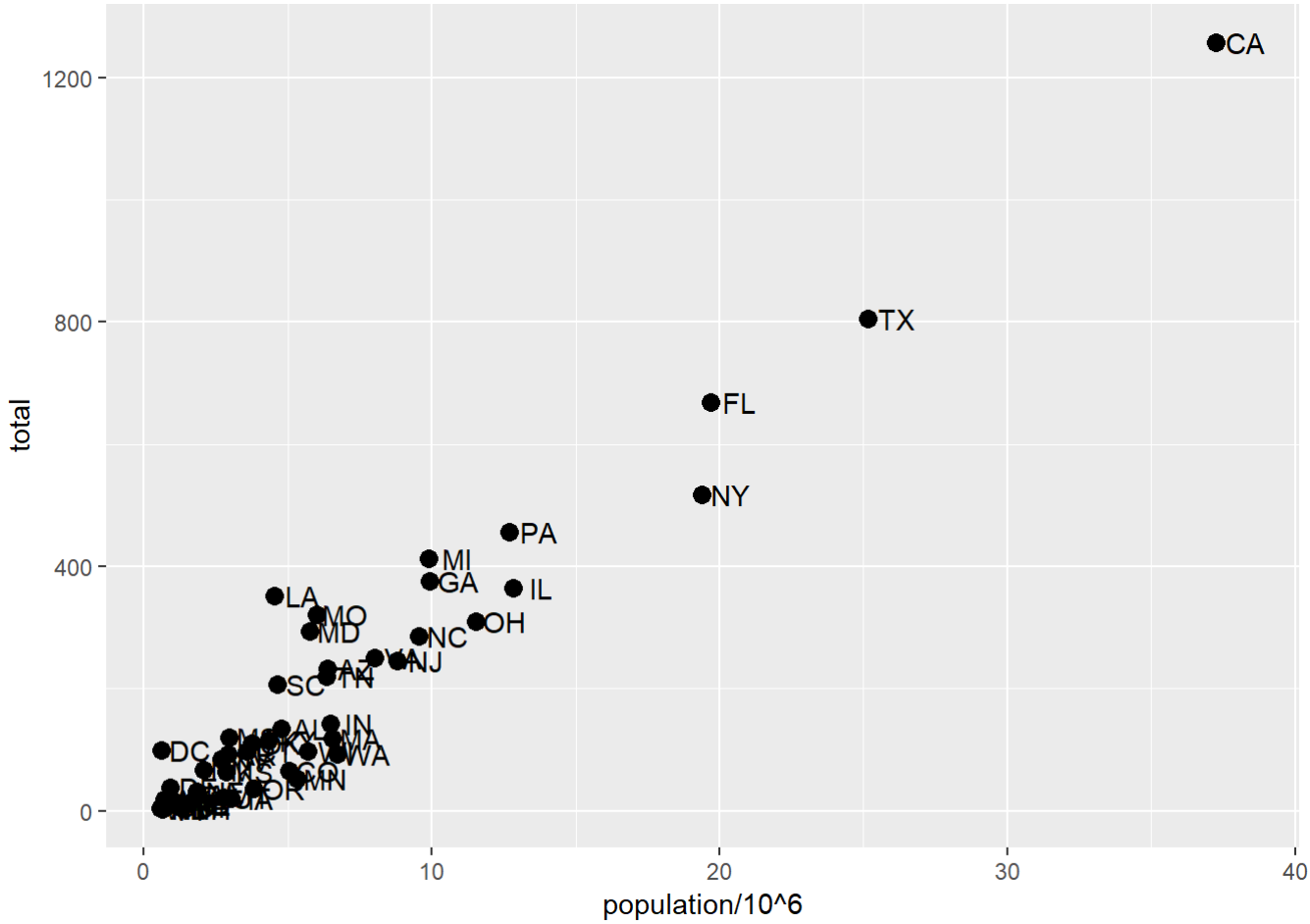


We put the size argument outside of aes. Size is not a mapping, it affects all the points the same and was outside of aes. We now need to shift the labels due to the points being larger.

```
?geom_text
```

The help file above shows that we can do so with the *nudge_x* argument.

```
p + geom_point(aes(population/10^6, total), size = 3) +
  geom_text(aes(population/10^6, total, label = abb), nudge_x = 1)
```



Nudge_x = 1 moves all geom_text labels slightly to the right, allowing us to read it.

So far, we have been quite inefficient by mapping population and total twice. Instead, we can avoid this by adding a global aesthetic mapping. This has to be done as we define the blank slate - at the *ggplot()* function, it has an argument for us to define aesthetic mappings.

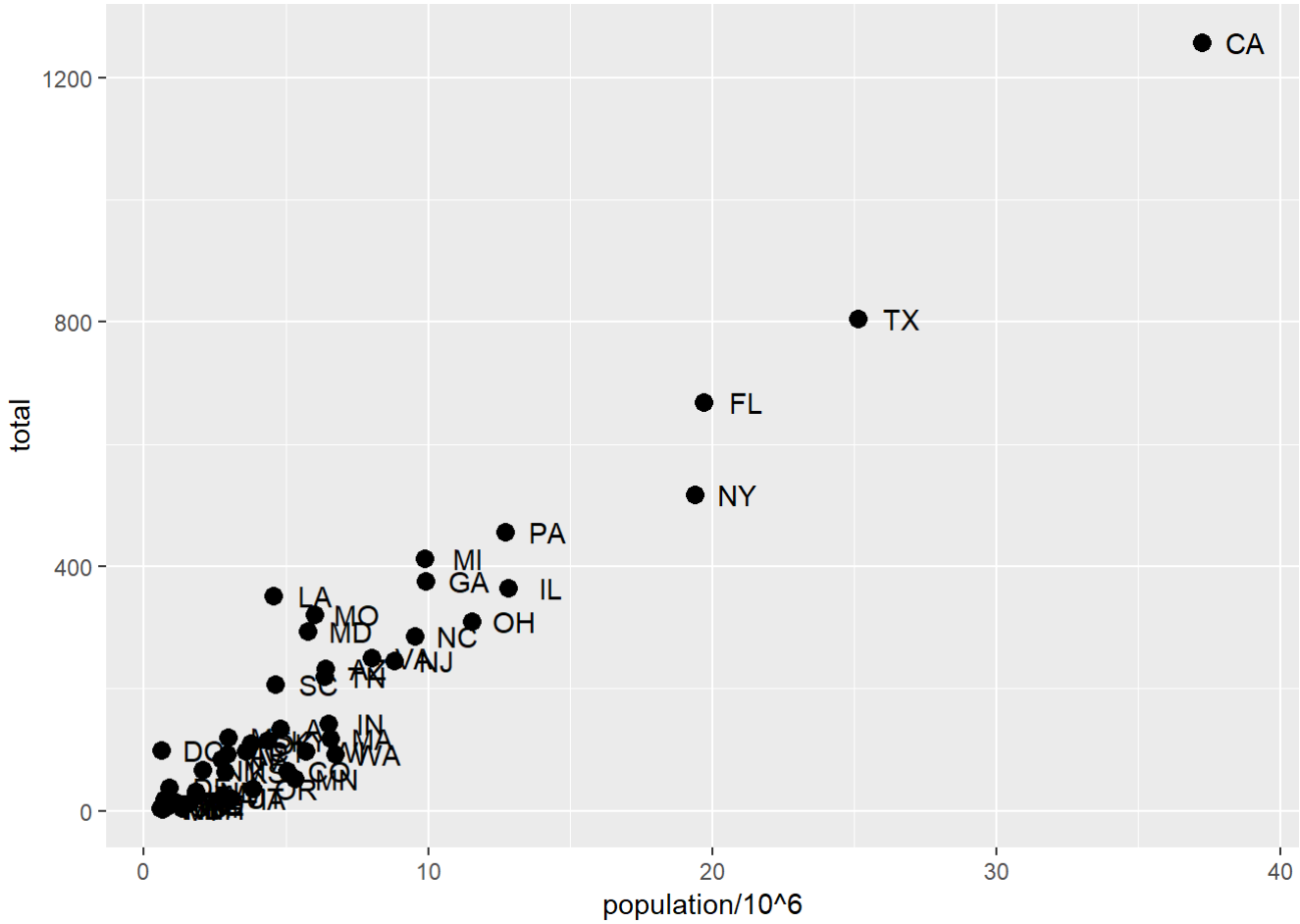
We can check all arguments a function has with the *args()* function.

```
args(ggplot)
```

```
## function (data = NULL, mapping = aes(), ..., environment = parent.frame())
## NULL
```

However, if we define a global aesthetic mapping in ggplot, then all geometries added as layers will default to this global.

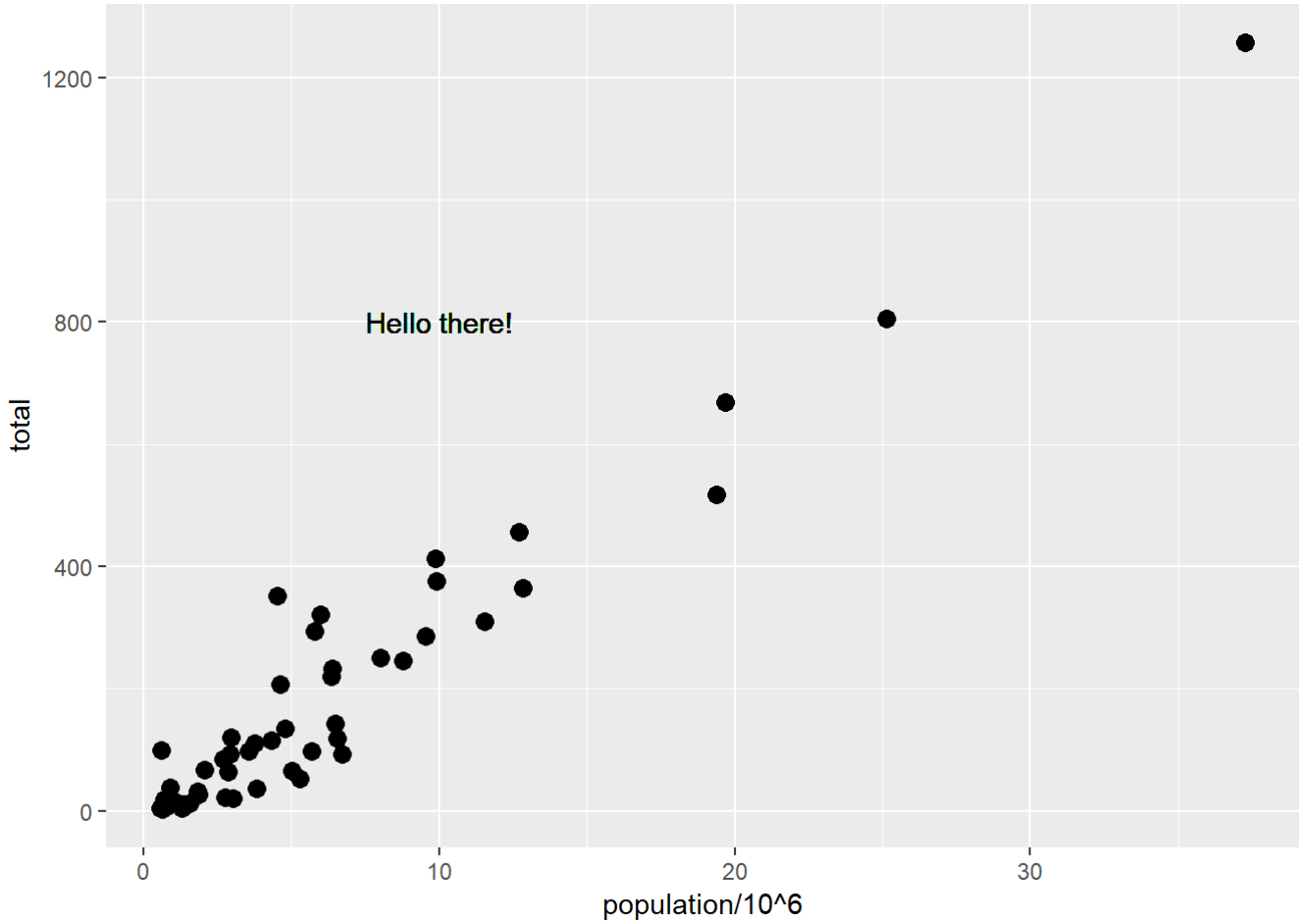
```
p <- murders %>% ggplot(aes(population/10^6, total, label = abb))
p + geom_point(size = 3) +
  geom_text(nudge_x = 1.5)
```



Note that the arguments kept in *geom_point()* and *geom_text()* are specific only to those two geometries and so must be kept there. The *geom_point()* did not need the label argument, so simply ignored it.

To override global mappings, we simply use local mappings to override this.

```
p + geom_point(size = 3) +
  geom_text(aes(x = 10, y = 800, label = "Hello there!"))
```



The plot produced above shows that the labels are no longer there, the local label argument is used instead.

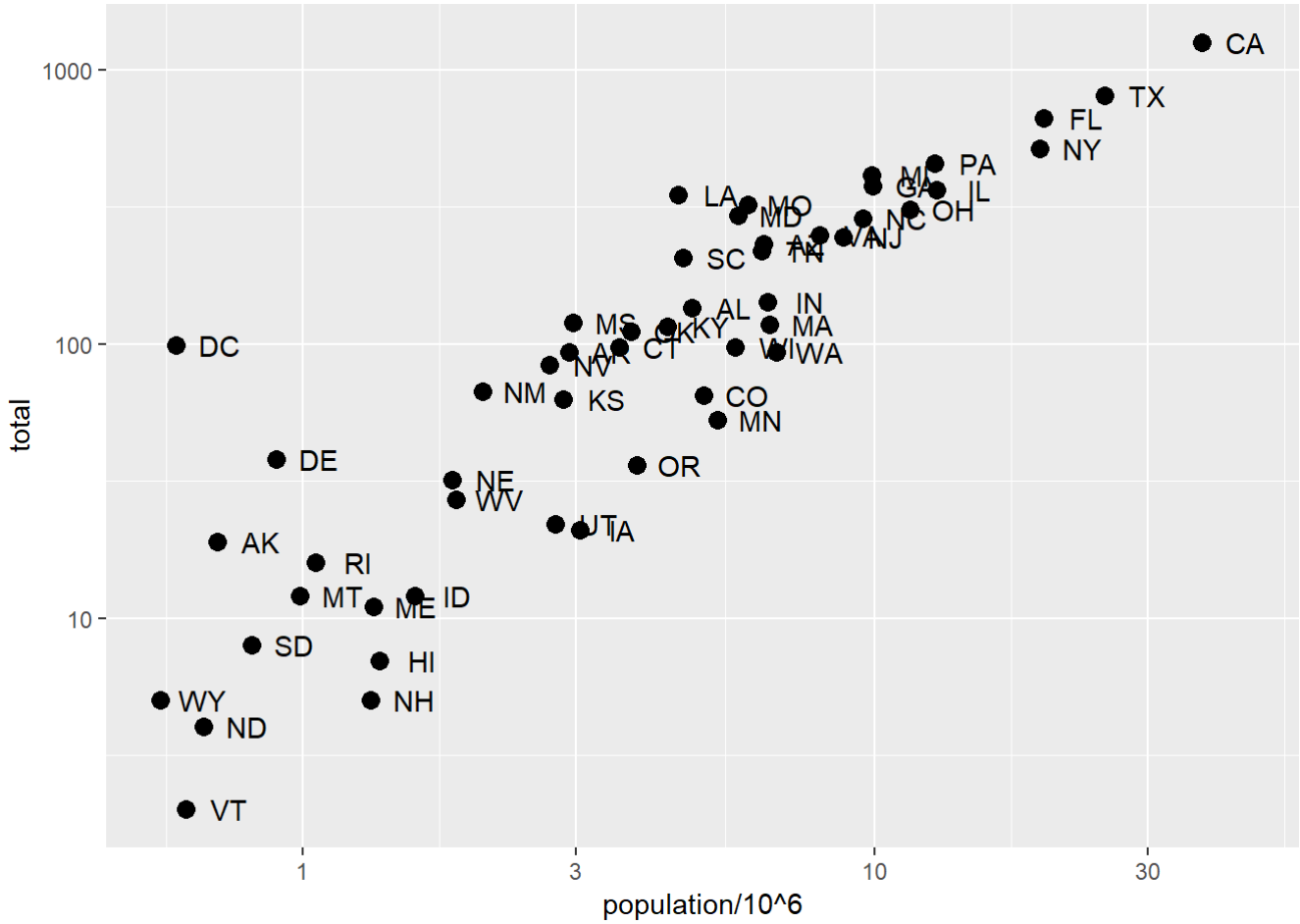
Scales, Labels, and Colors

To change our scales to the log scales for our example, we need to use the scales layer. We can use the *scales_x_continuous()* function to edit the behavior of scales.

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

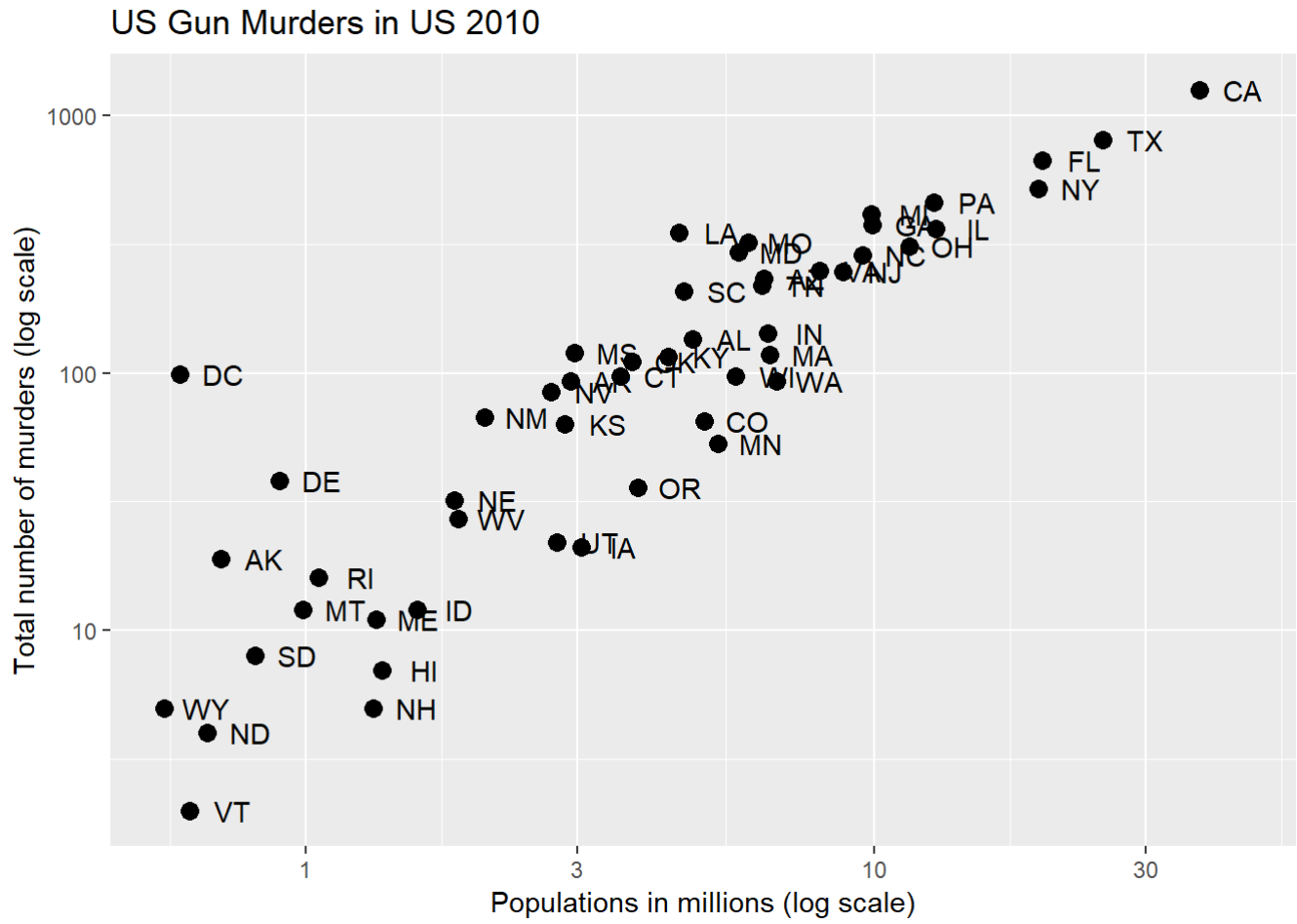
Notice that the nudge must be smaller due to a change in scale. The log transformation is also common enough that ggplot provides specialized functions.

```
p + geom_point(size = 3) +
  geom_text(nudge_x = 0.075) +
  scale_x_log10() +
  scale_y_log10()
```



We can use *xlab()* and *ylab()* functions to add labels to the respective axes, and *ggtitle()* to add a title to our plot.

```
p + geom_point(size = 3) +
  geom_text(nudge_x = 0.075) +
  scale_x_log10() +
  scale_y_log10() +
  xlab("Populations in millions (log scale)") +
  ylab("Total number of murders (log scale)") +
  ggtitle("US Gun Murders in US 2010")
```



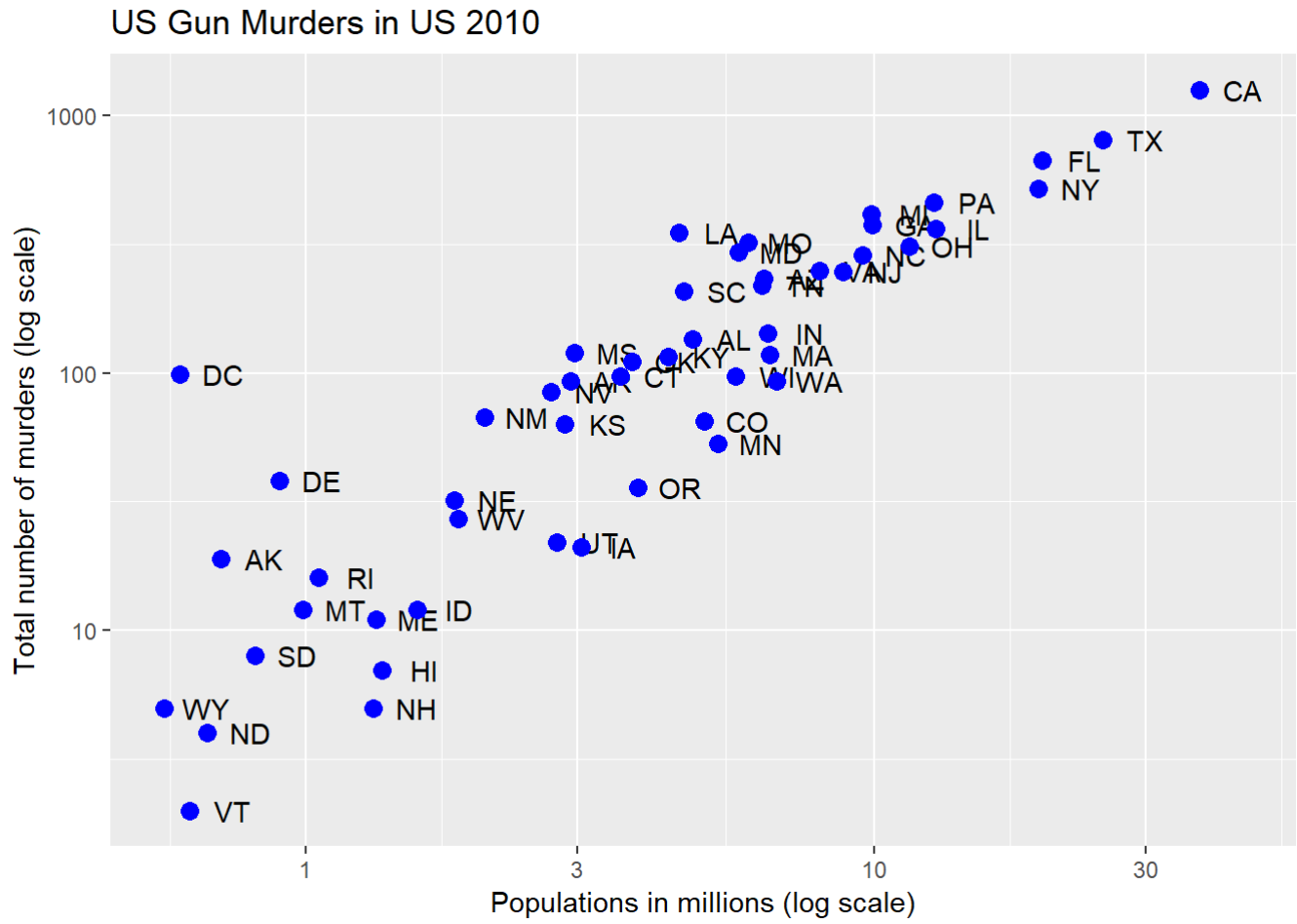
We can change the color of our points in the plot using the “color” argument in the *geom_point()* function.

In order to slowly try out the effects of changing color on our plot, we can save what we have done so far by assigning it to p.

```
p <- murders %>% ggplot(aes(population/10^6, total, label = abb)) +
  geom_text(nudge_x = 0.075) +
  scale_x_log10() +
  scale_y_log10() +
  xlab("Populations in millions (log scale)") +
  ylab("Total number of murders (log scale)") +
  ggtitle("US Gun Murders in US 2010")
```

To make our points blue, we use “blue” for the color argument.

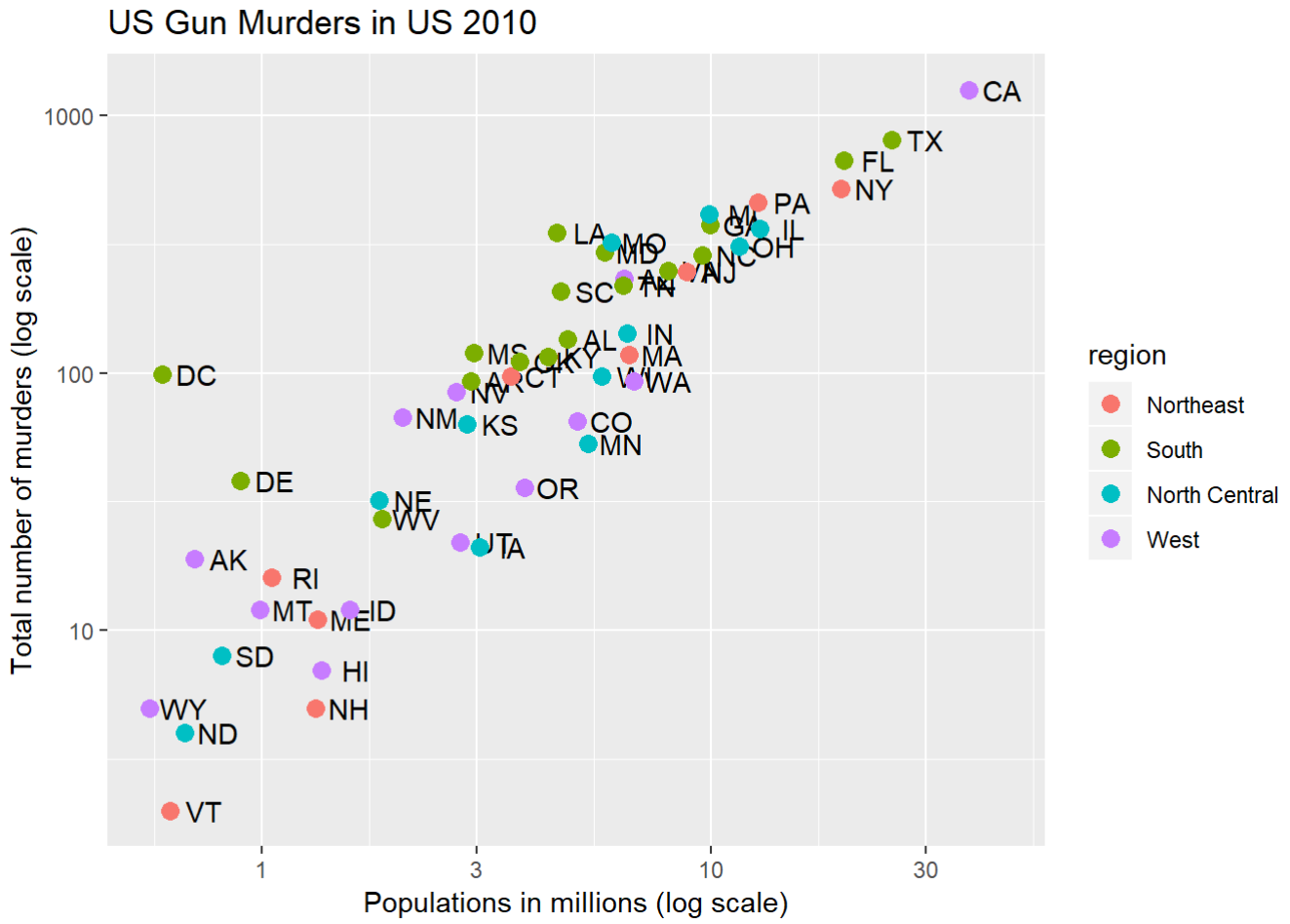
```
p + geom_point(size = 3, color = "blue")
```



However, we want the color to be associated with geographical regions. By default, if we assign a categorical variable to the color argument, it will automatically assign a different color for each category and also adds a legend.

We will need to use mapping, so we have to use aes.

```
p + geom_point(aes(col=region), size = 3)
```



Notice that the “col” argument is inside aes, while size is still outside. The x & y mapping was already defined earlier in the global aesthetic mapping and so do not have to be repeated here.

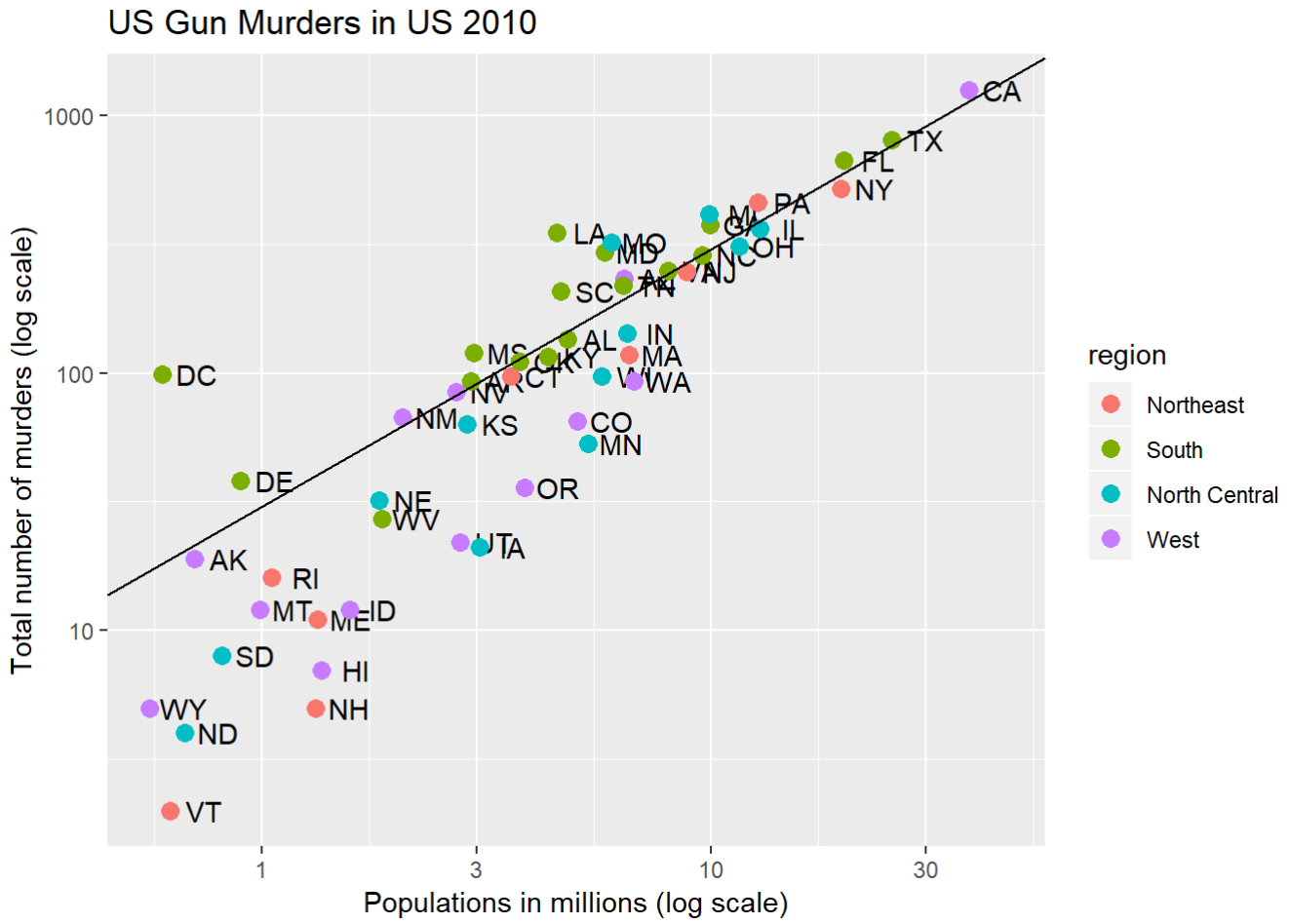
We also want to add a line that represents the average murder rate in the entire country. Once we determine the per million rate to be “r”, the line will be y = r*x.

We will make use of dplyr to find the murder rate.

```
library(dplyr)
r <- murders %>%
  summarize(rate = sum(total) / sum(population) * 10^6) %>% .$rate
```

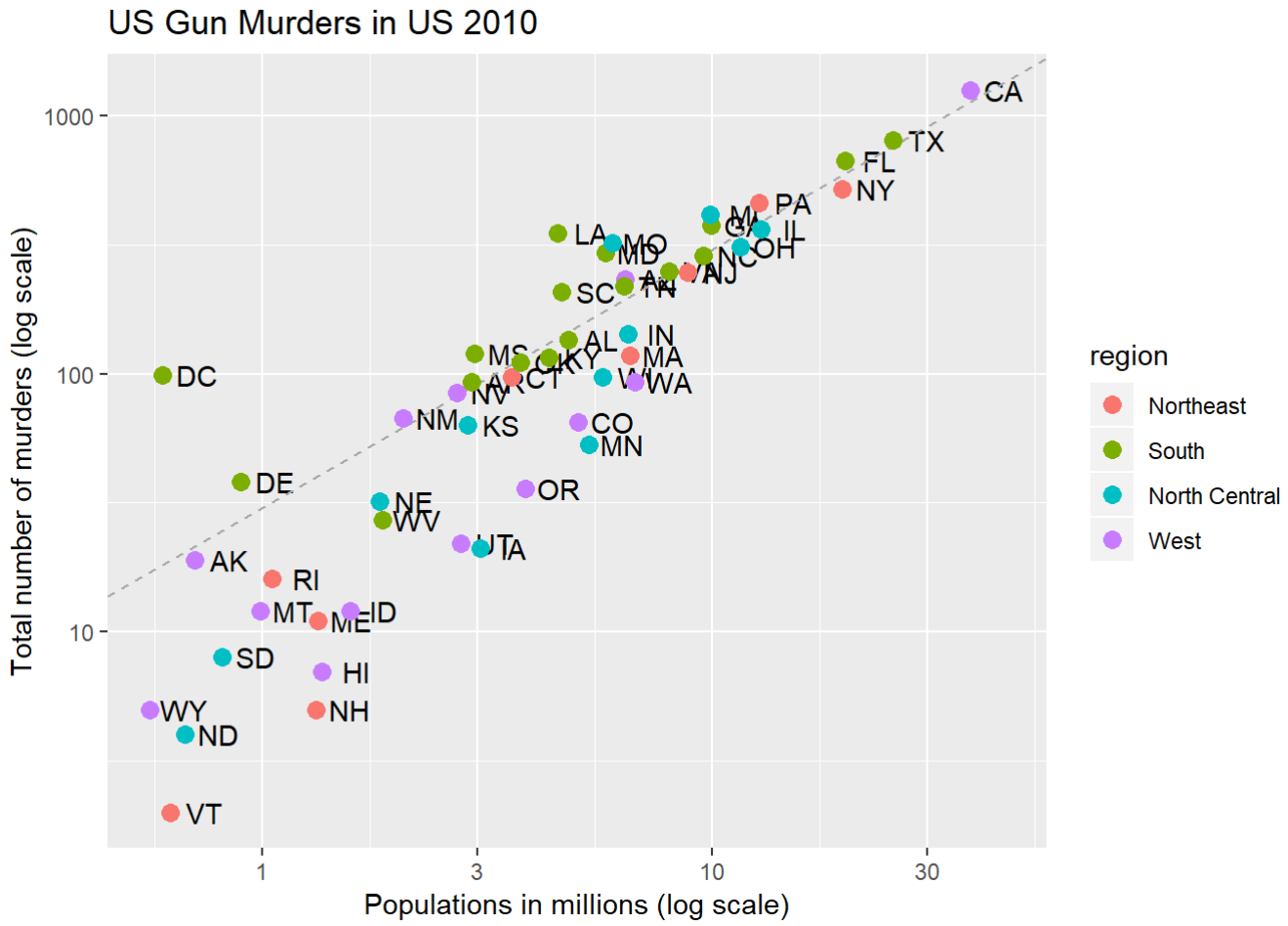
To add a line, we use the *geom_abline()* function. Ggplot uses ‘ab’ in the name to remind us we’re supplying the intercept a and slope b. The default line for *geom_abline()* has slope 1, intercept 0. This means we only have to specify the intercept.

```
p + geom_point(aes(col=region), size = 3) +
  geom_abline(intercept = log10(r))
```



To recreate the original plot, we have to change the line type from solid to dashed, change the color from black to grey, and also, we need to draw the line before the points. This is so that the line will be in the background relative to the points. 'lty' = 2 changes the line type to dashed and 'color' for abline is for the line itself.

```
p <- p +
  geom_abline(intercept = log10(r), lty = 2, color = "darkgrey") +
  geom_point(aes(col=region), size = 3)
p
```

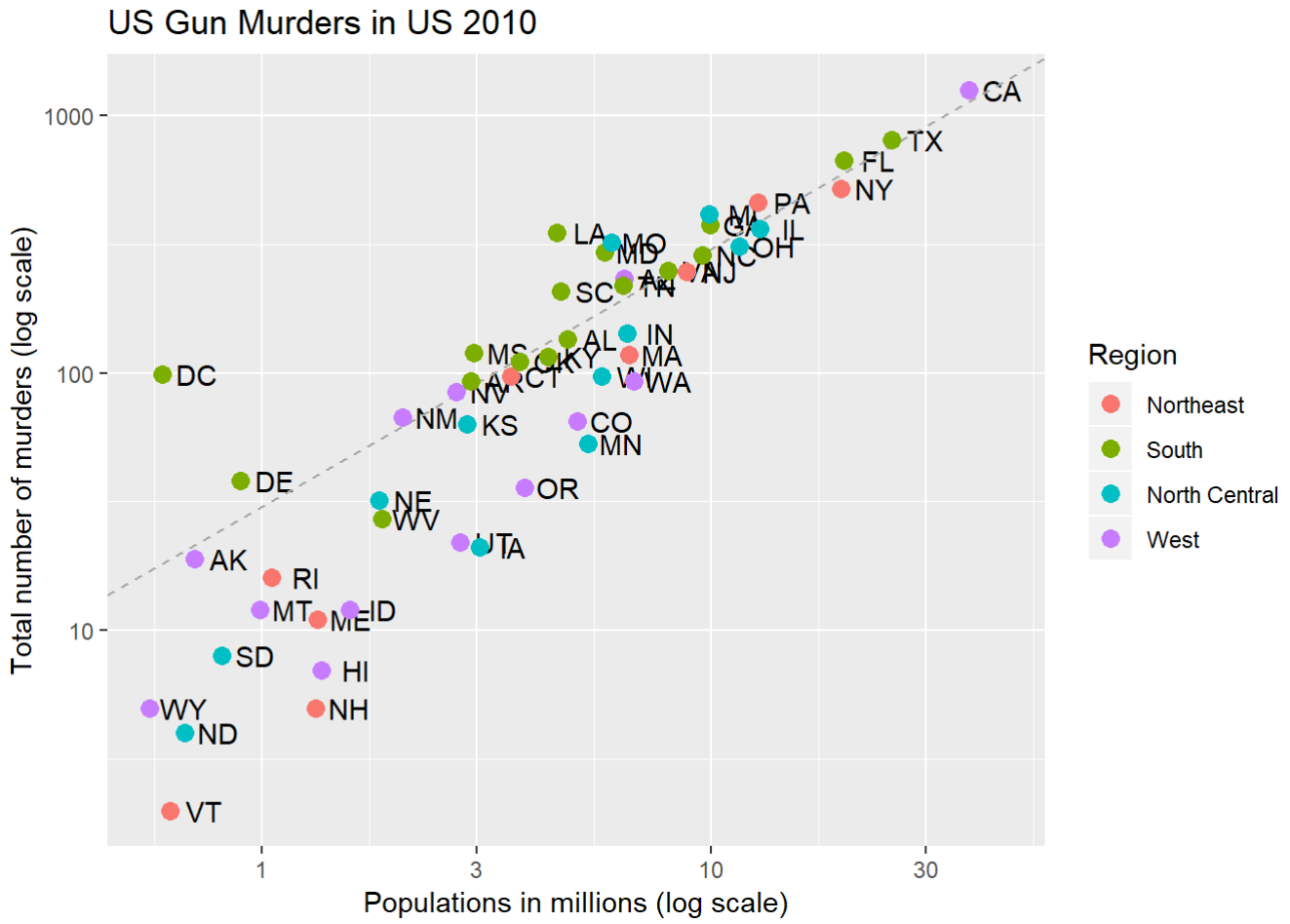


Also, notice that abline is now before point.

Ggplot is very flexible that any small changes that you might want to make are always possible. One such small change is to capitalize the word region in the legend.

We can do this with *scale_color_discrete()* function.

```
p <- p + scale_color_discrete(name = "Region")
p
```



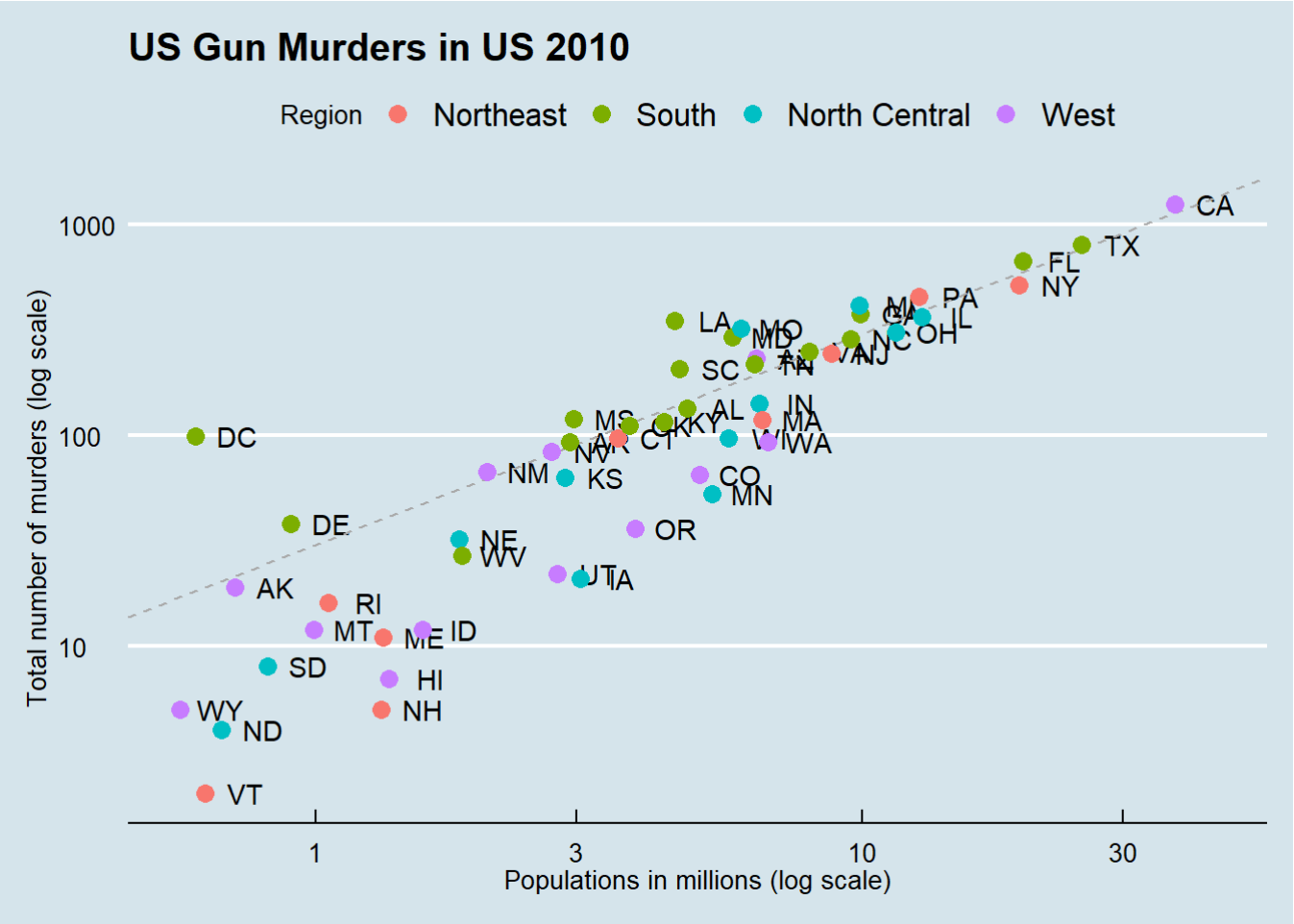
Add On Packages

Ggplot allows us to use additional add on packages in order to further extend its functionality. We will be exploring the *ggthemes* and *ggrepel* add on packages in order to duplicate the example graph.

The style of a ggplot graph can be changed using the theme function. Several themes are included as part of the ggplot2 package.

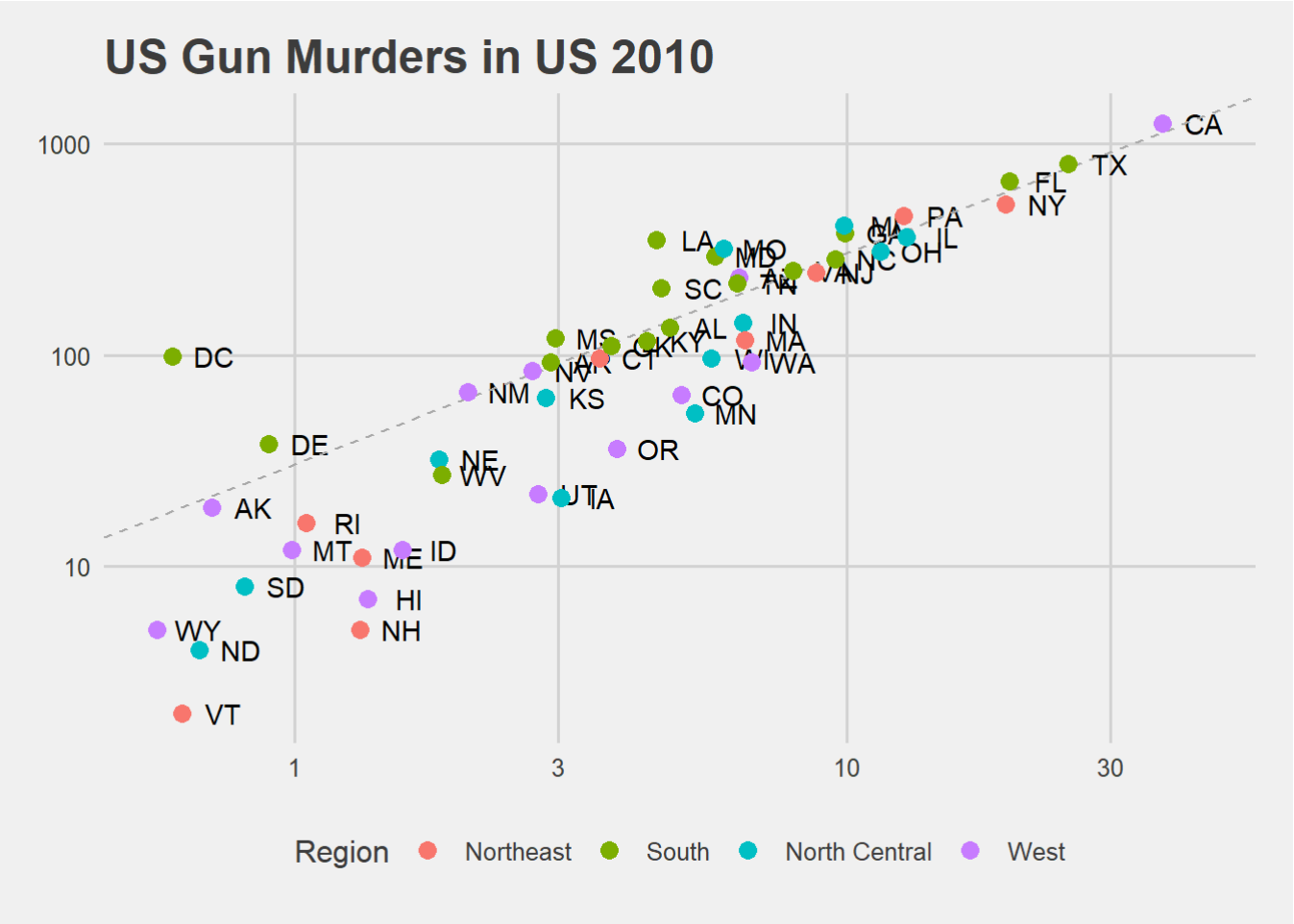
We will be installing and loading *ggthemes* and then using the *theme_economist()*.

```
library(ggthemes)
p + theme_economist()
```



Simply change the trailing function from this point to try out other themes.

```
p + theme_fivethirtyeight()
```

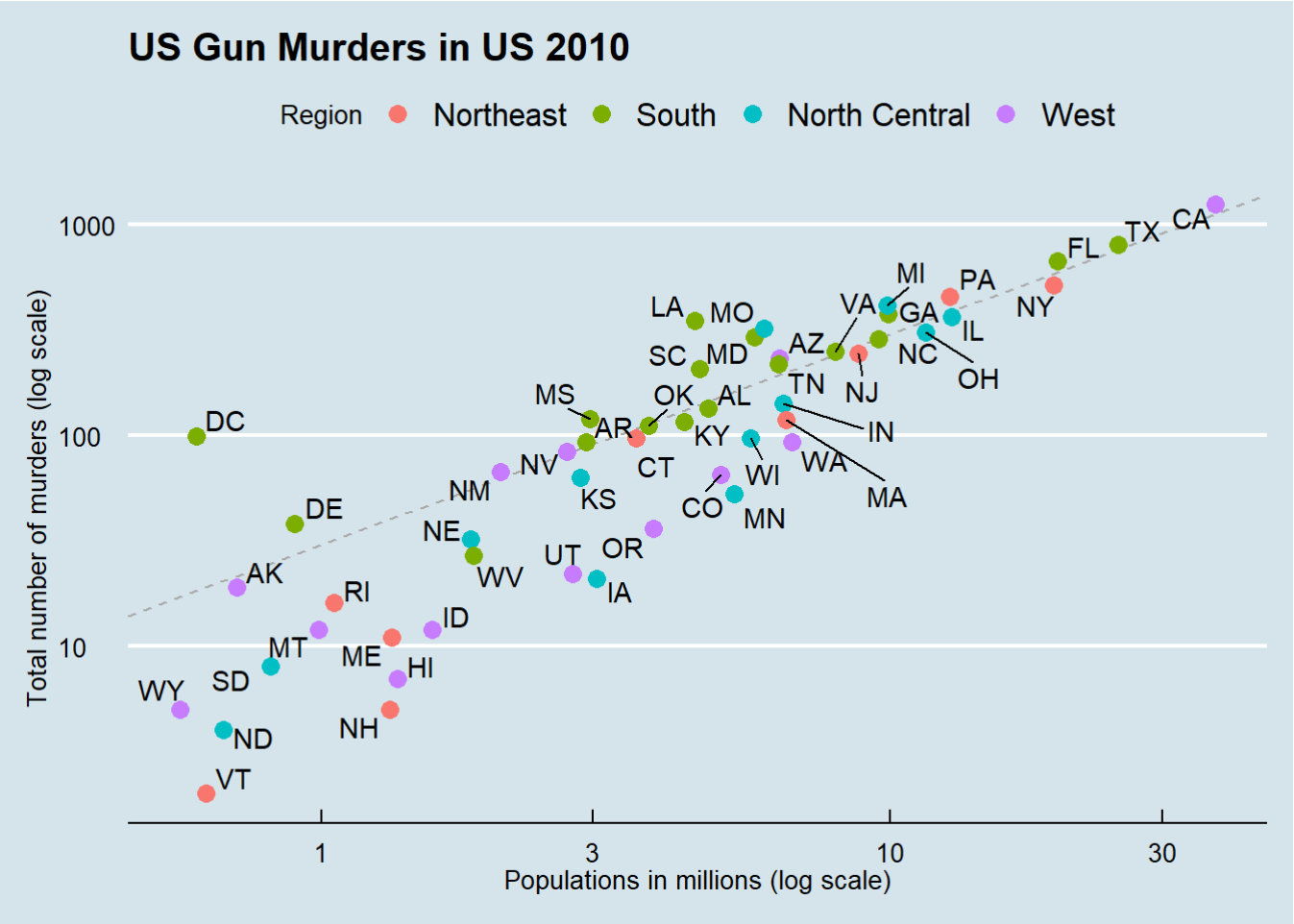


Finally, we just want to reposition our labels. Currently, some of the labels overlap, making it hard to read.

This is where *ggrepel* comes in, it can add geometry for labels to ensure this behavior is avoided. We simply need to change the *geom_text()* layer a *geom_text_repel()* layer instead.

From scratch, these are **all steps** in the correct order.

```
library(ggthemes)
library(ggrepel)
### First define the slope of the line
r <- murders %>%
  summarize(rate = sum(total) / sum(population) * 10^6) %>% .$rate
## Now make the plot
murders %>% ggplot(aes(population/10^6, total, label = abb)) +
  geom_abline(intercept = log10(r), lty = 2, color = "darkgrey") +
  geom_point(aes(col=region), size = 3) +
  geom_text_repel() +
  scale_x_log10() +
  scale_y_log10() +
  xlab("Populations in millions (log scale)") +
  ylab("Total number of murders (log scale)") +
  ggtitle("US Gun Murders in US 2010") +
  scale_color_discrete(name = "Region") +
  theme_economist()
```



Other Examples

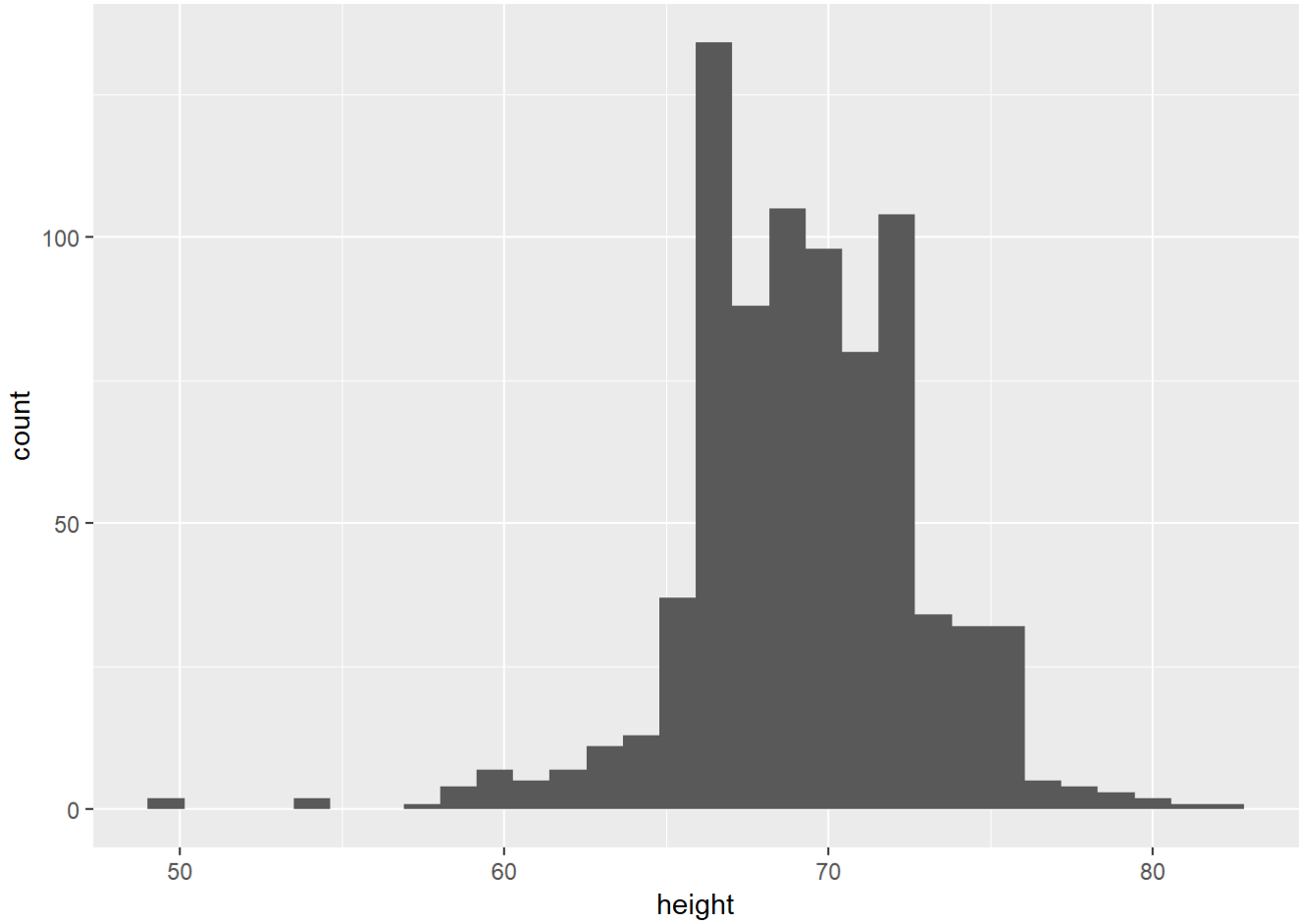
Now that we know ggplot, we can try to make some of the summary plots we have previously described, starting with the histogram.

```
library(dslabs)
data(heights)
#heights %>% filter(sex=="Male") try this line yourself
```

Now, we will be using *geom_histogram()*, notice the naming consistency between ggplot functions. From the help file, we can see that only the value x is needed to construct a histogram.

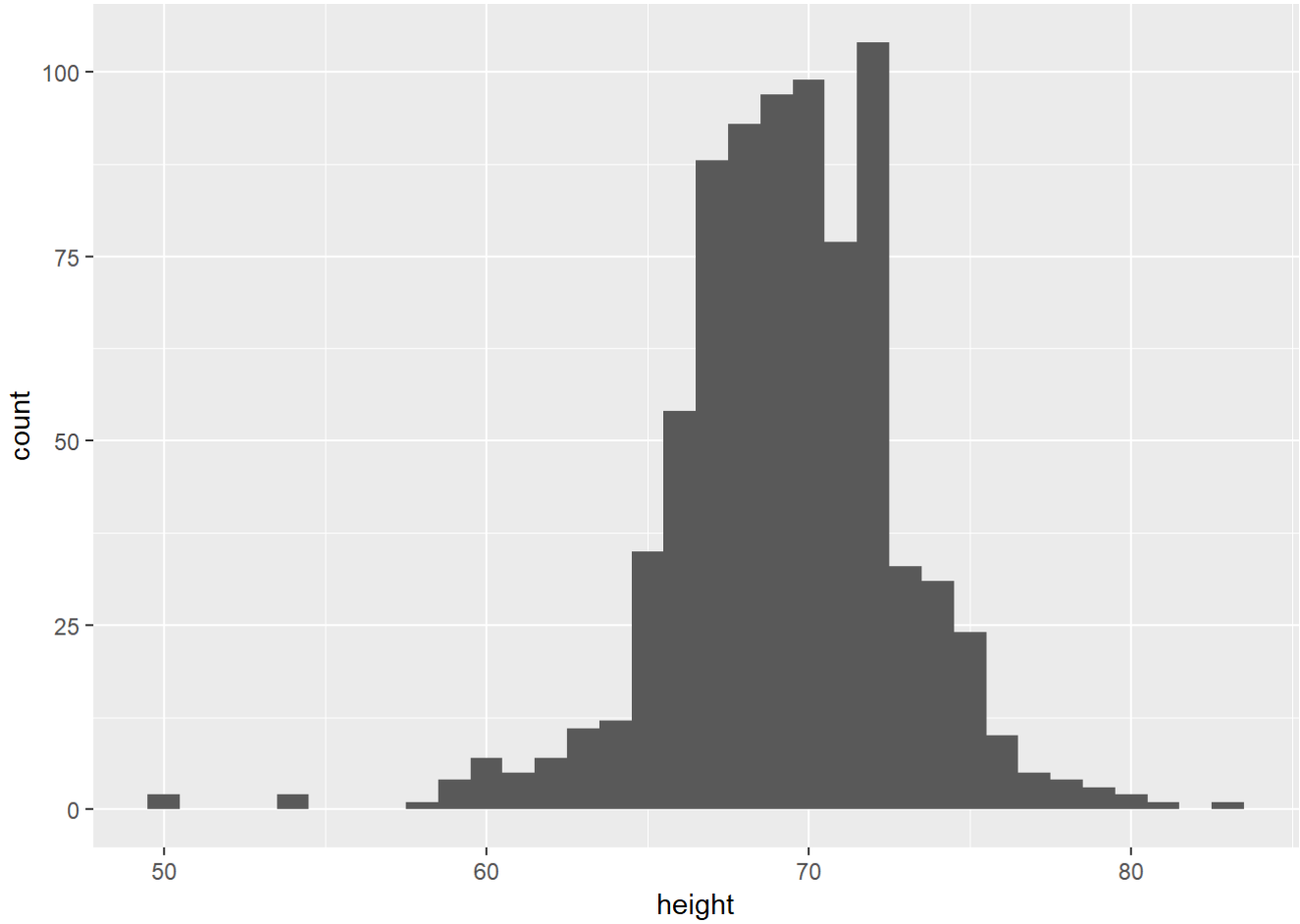
```
p <- heights %>%
  filter(sex=="Male") %>%
  ggplot(aes(x = height))
p + geom_histogram()
```

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



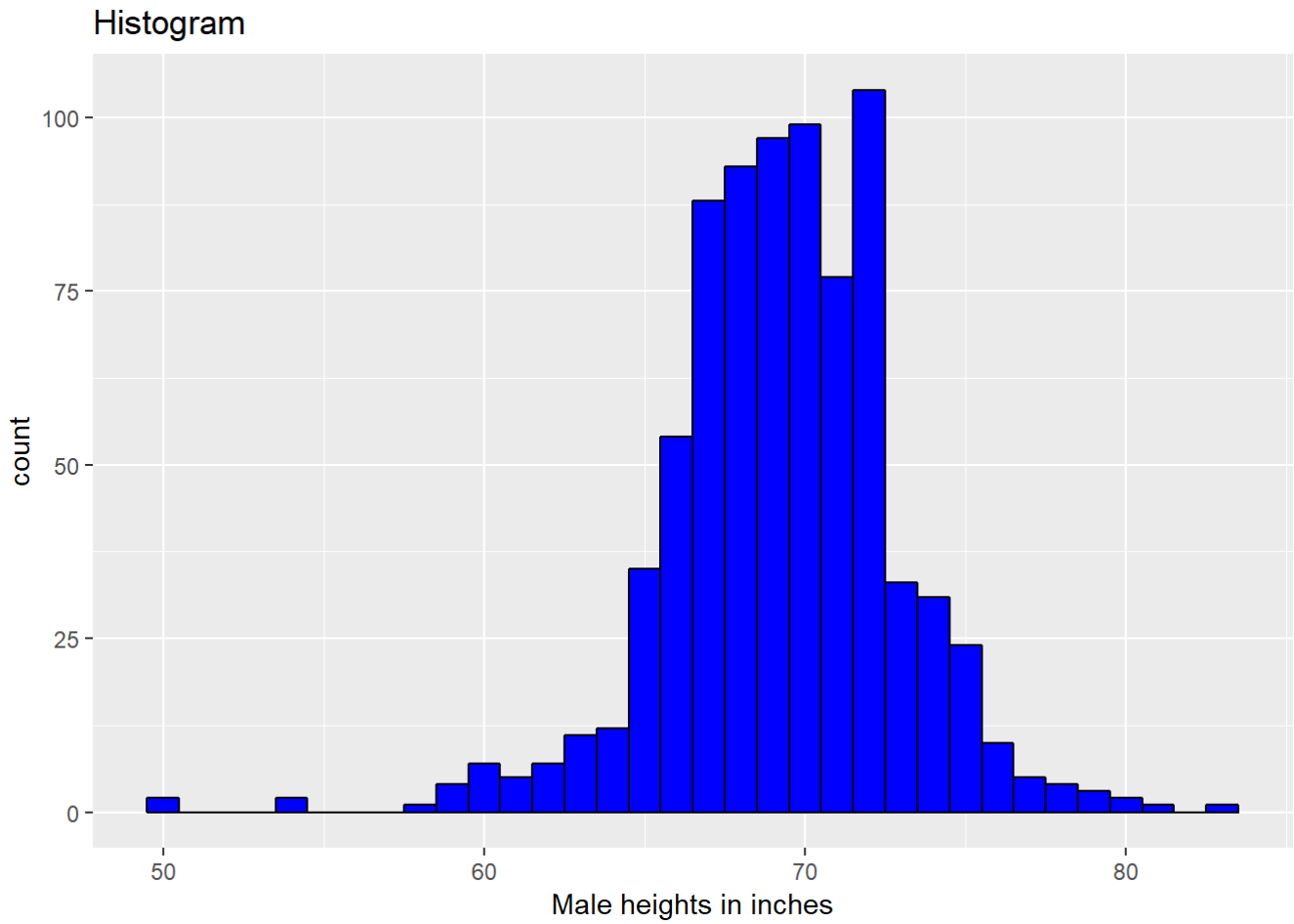
Notice that we get a warning telling us that as bin width was not specified it was selected for us. Here we will use binwidth = 1.

```
p + geom_histogram(binwidth = 1)
```



To display the flexibility of ggplot, we will change the colors and add a title.

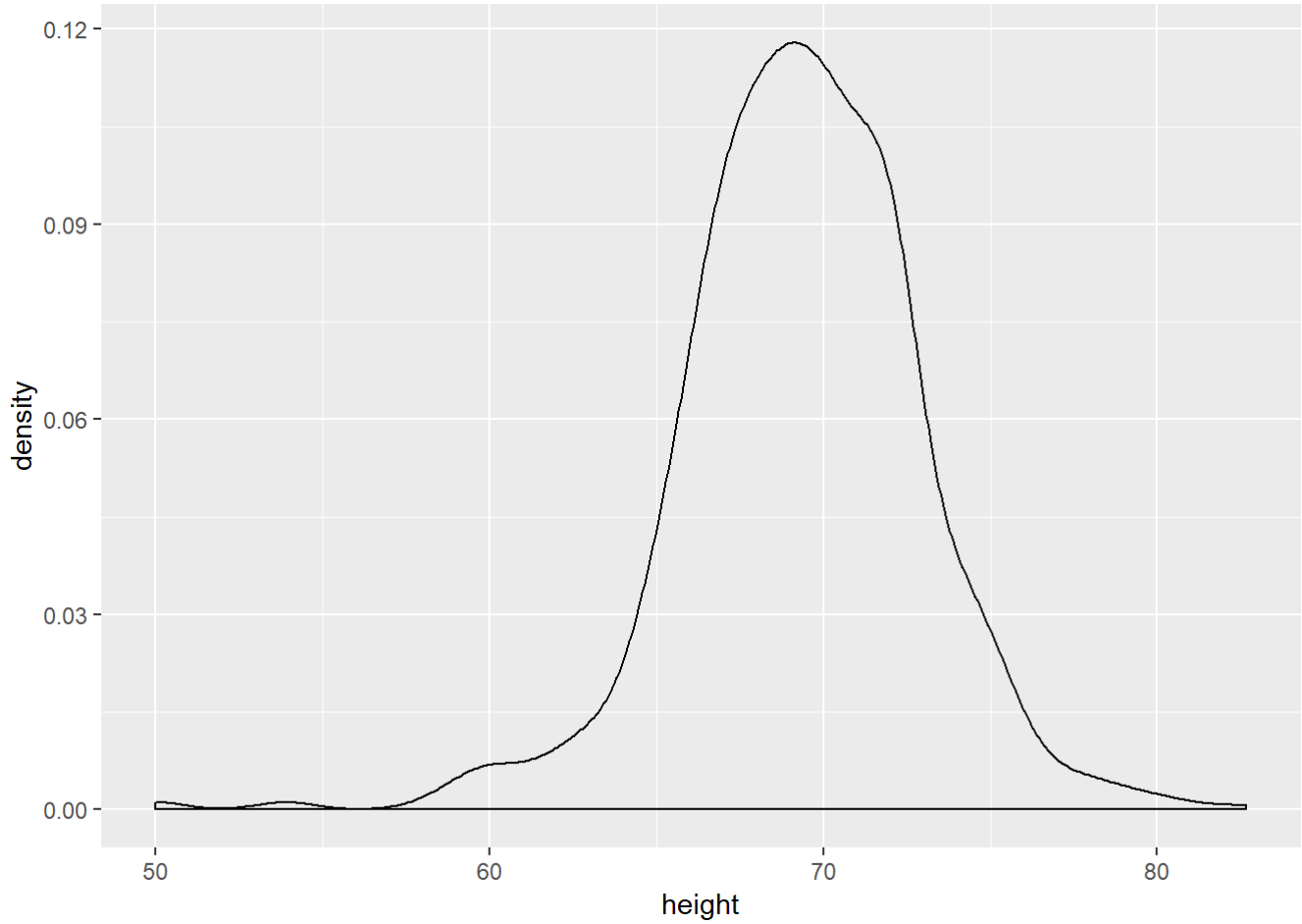
```
p + geom_histogram(binwidth = 1, fill = "blue", col = "black") +
  xlab("Male heights in inches") +
  ggtitle("Histogram")
```



Note that 'fill' is for the bar itself, while 'col' is for the outline.

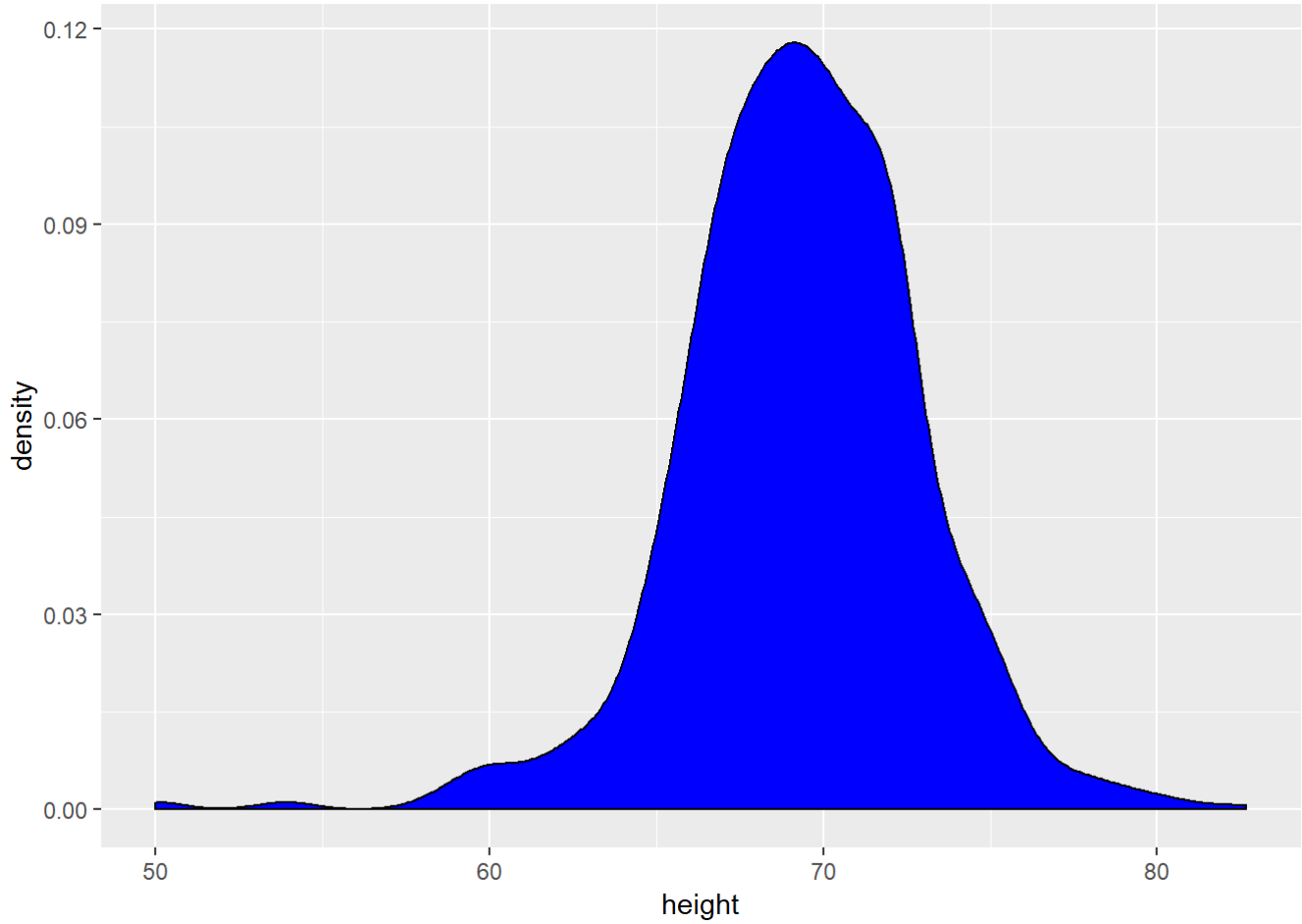
We can also create smooth densities with the *geom_density()* function. Instead of *geom_histogram()* above, we use this instead.

```
p + geom_density()
```



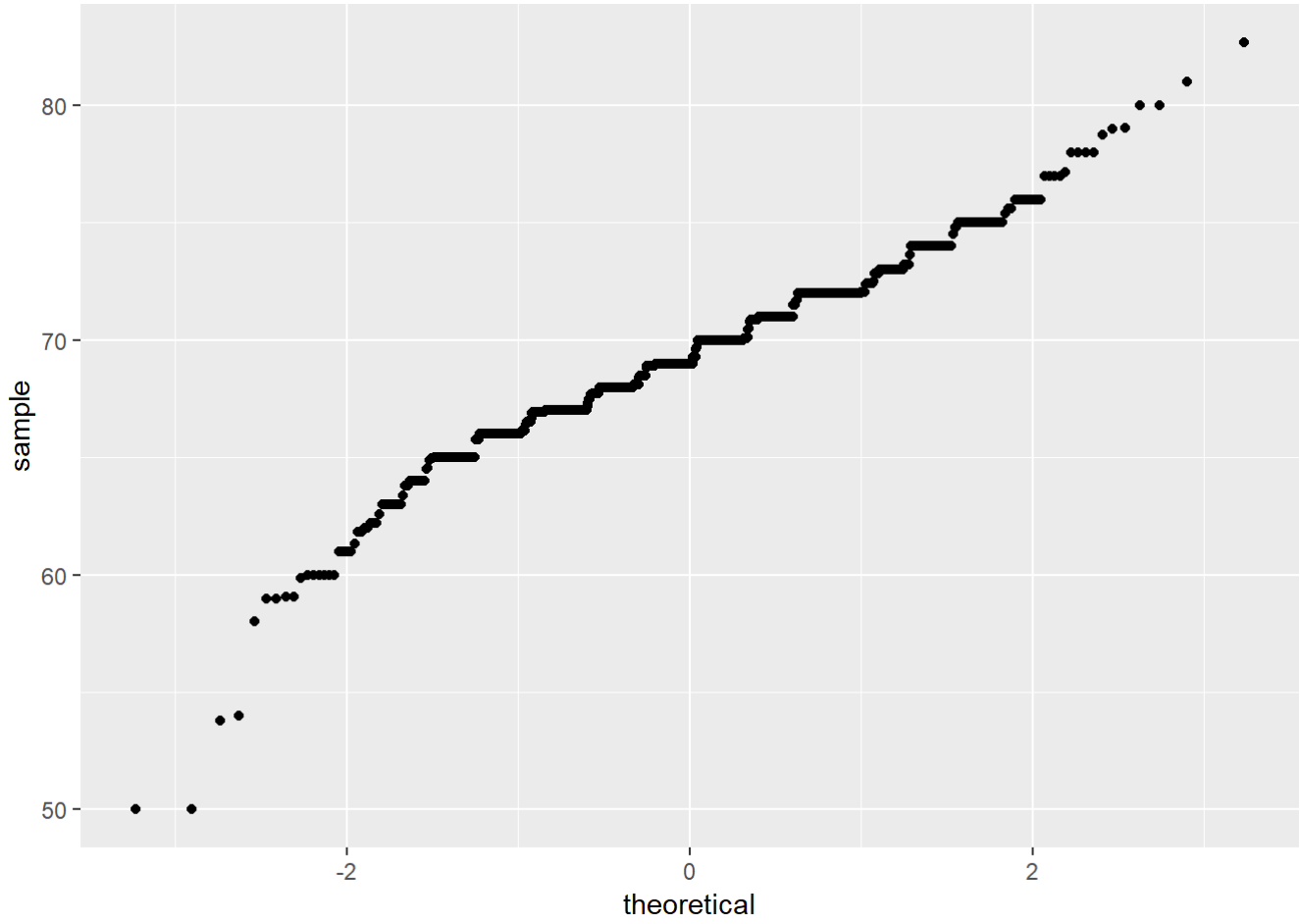
To add color, we simply use the 'fill' argument.

```
p + geom_density(fill = "blue")
```



For q-q plots, we make use of the *geom_qq()* geometry. From the help file, we learn that we need to specify the sample argument. Instead of x, it now takes in sample. Therefore, we need to change our p.

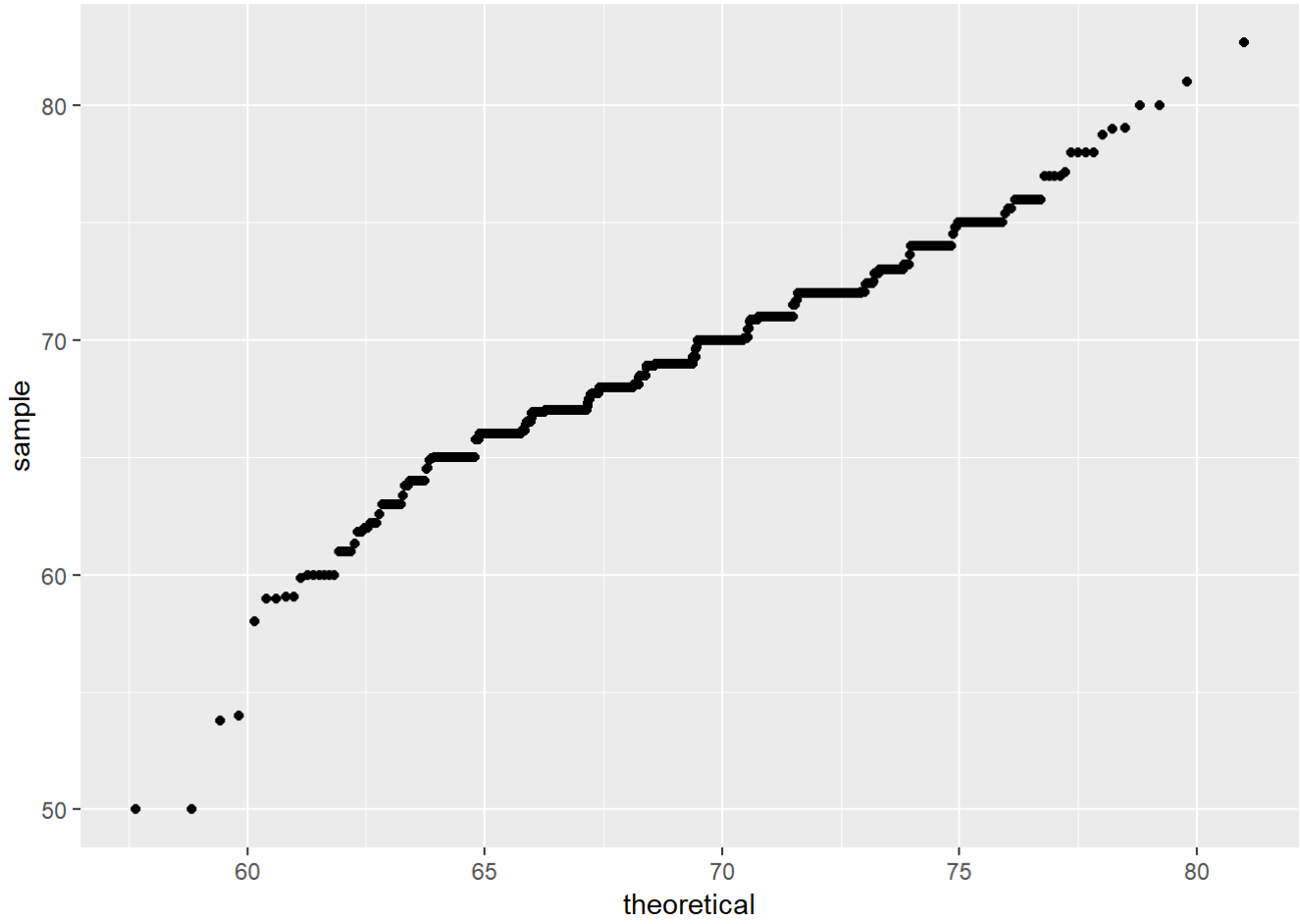
```
p <- heights %>%
  filter(sex=="Male") %>%
  ggplot(aes(sample = height))
p + geom_qq()
```



Note that by default the q-q plot is compared to the normal distribution with average = 0 and standard deviation = 1. The standard normal distribution.

From the help file, we know that we can change this with the dparams argument. Now we just need to define an object 'params' that will have the mean and standard deviation of our data using some dplyr functions.

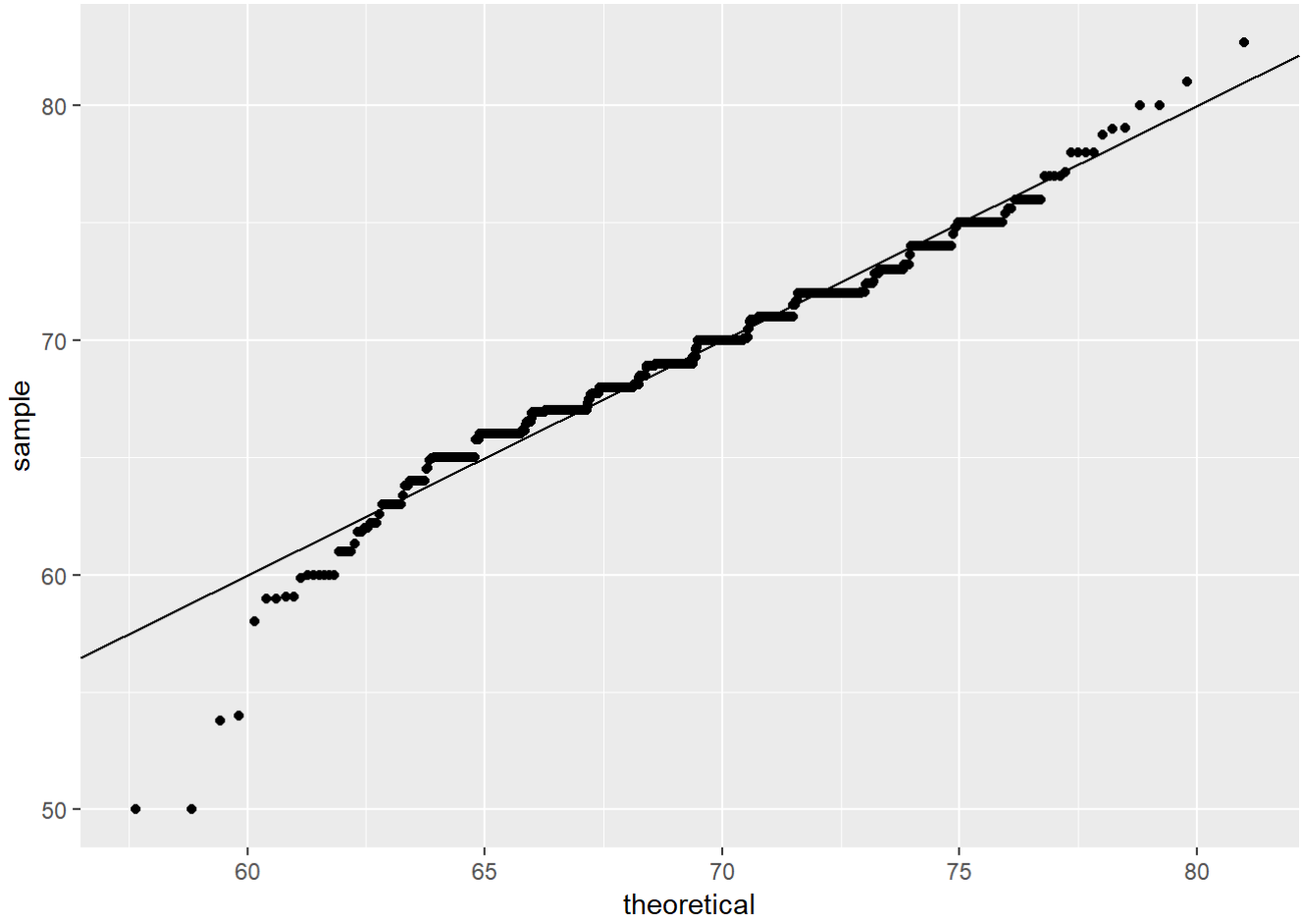
```
params <- heights %>%
  filter(sex == "Male") %>%
  summarise(mean = mean(height), sd = sd(height))
p + geom_qq(dparams = params)
```



Now the q-q plot is plotted against a normal distribution with the same mean and standard deviation as our data.

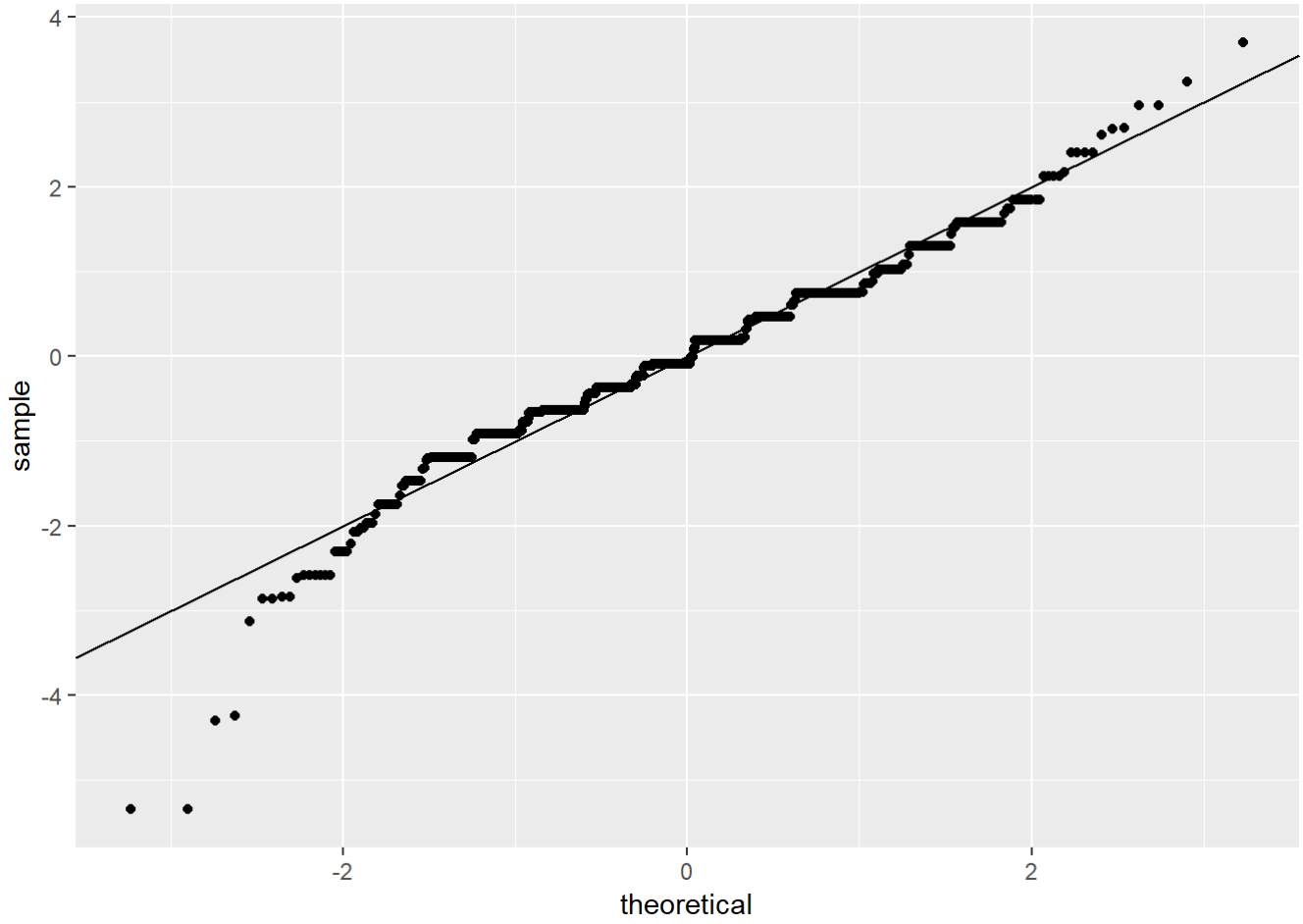
We can now add identity lines to see just how well the normal approximation works. This is done with the *geom_abline()* layer.

```
p + geom_qq(dparams = params) +
  geom_abline()
```



Alternatively, we can scale the data in standard units before plotting it against the standard normal distribution, saving us the step of having to compute the mean and standard deviation of our dataset.

```
heights %>% filter(sex=="Male") %>%
  ggplot(aes(sample = scale(height))) +
  geom_qq() +
  geom_abline()
```



Notice that the code looks a little cleaner.

We can also make grids of plots. We can use the *gridExtra* package, which has a function *grid.arrange()*

Firstly, we must define and assign each plot to an object.

```
p <- heights %>% filter(sex=="Male") %>% ggplot(aes(x = height))
p1 <- p + geom_histogram(binwidth = 1, fill = "blue", col = "black")
p2 <- p + geom_histogram(binwidth = 2, fill = "blue", col = "black")
p3 <- p + geom_histogram(binwidth = 3, fill = "blue", col = "black")
```

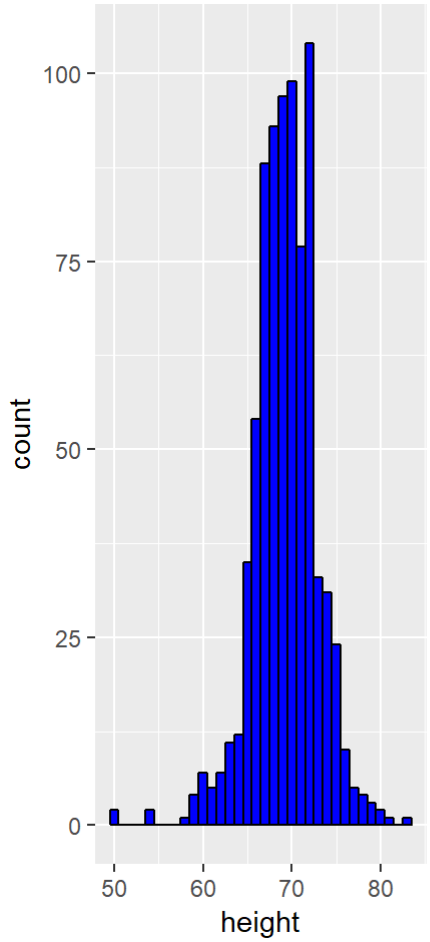
Then, to show them next to each other, we use the function *grid.arrange()*.

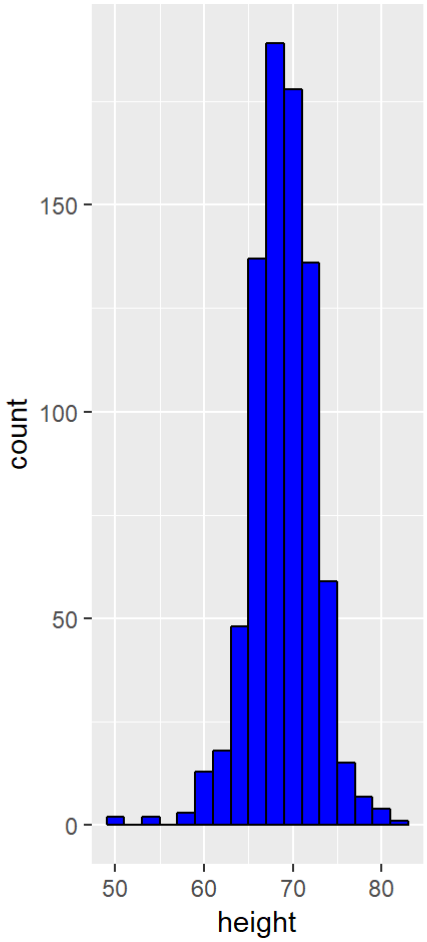
```
library(gridExtra)
```

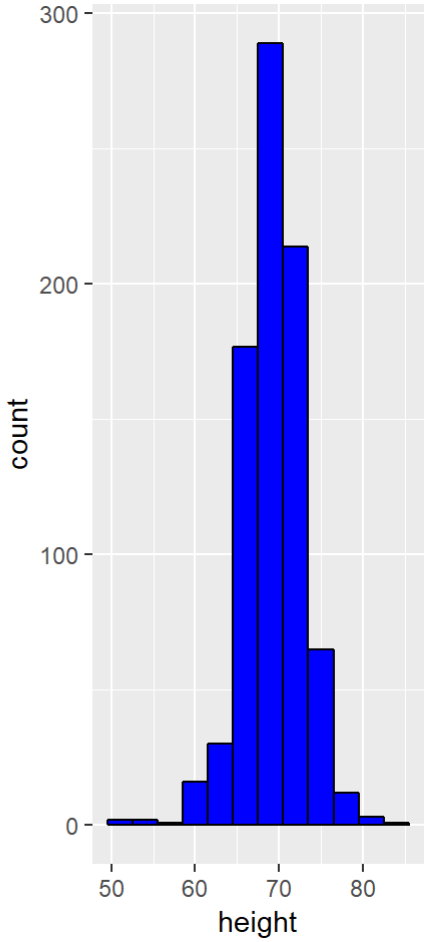
```
##
## Attaching package: 'gridExtra'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
grid.arrange(p1,p2,p3, ncol = 3)
```


count
height
0
25
50
75
100
50
60
70
80


count
height
0
50
100
150
50
60
70
80


count
height
0
100
200
300
50
60
70
80

Advanced Data Visualisation

Dr. Liu Qizhang

1 March 2019

Introduction

This file covers advanced data exploration, such as visualizing spatial data and data transformation. We will also cover the ggmap and leaflet packages.

Grammar of Graphics

Data - We always start with specifying the data source of the graph.

Aes - Data mapping to confirm the axes based on data dimensions and positions of various data points in the plot.

It also allows us to specify the form of encodings, such as color, size, shape and so on.

Geom - Geometric objects in which we would like our data to be depicted. Such as points, lines, bars, etc.

Coord(“Co-ord”) - The coordinate system that the visualization is based on. Cartesian or polar.

Scale - The specific scale needed to represent multiple values or a range.

Facet - To create subplots based on specific data dimensions.

Stat - Statistical measures to apply to the data before it is plotted.

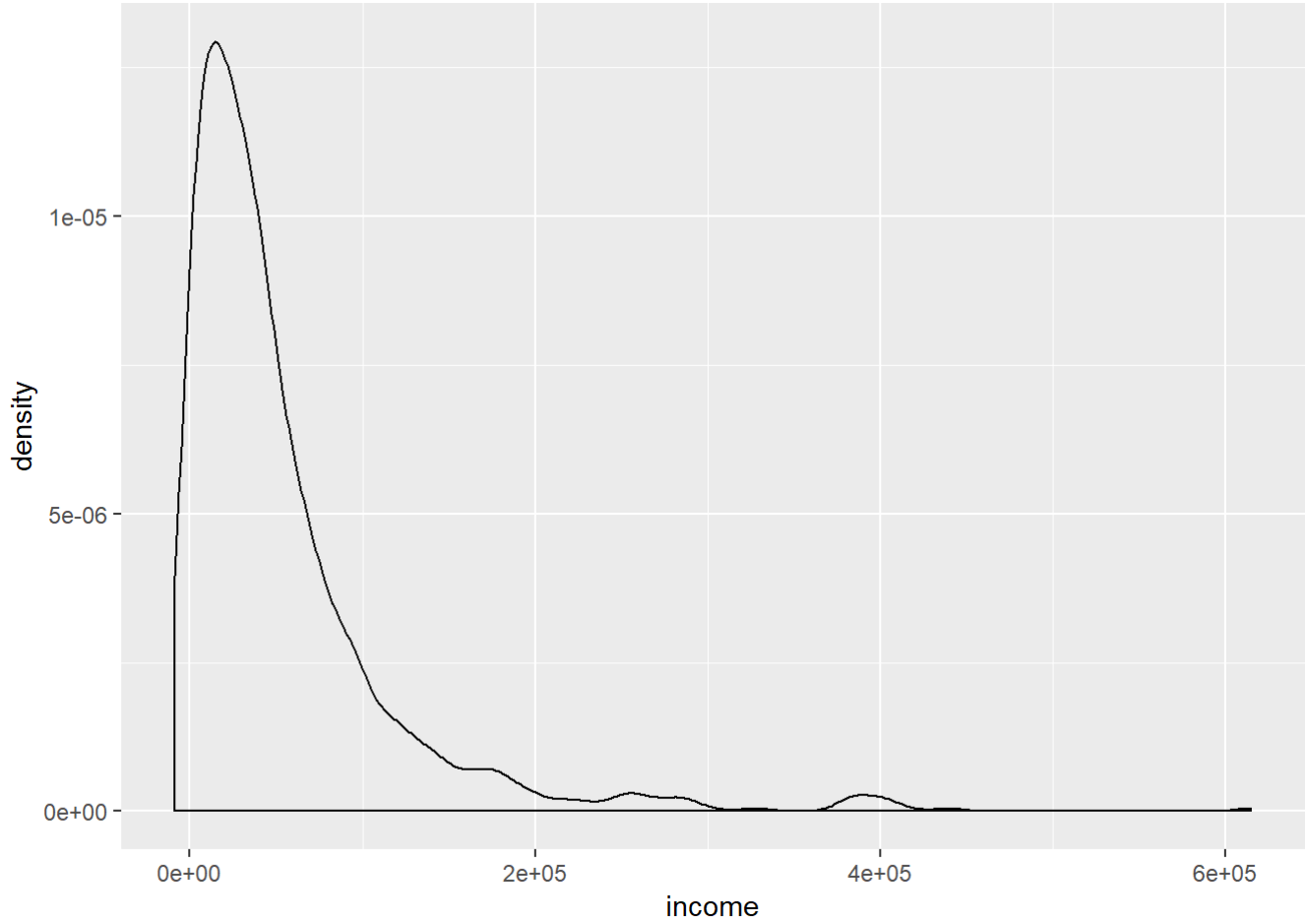
Position - How your geometric objects are positioned.

To load the ‘custdata’ dataset

```
custdata <- read.table('custdata.tsv', header=T, sep='\t')
```

Here is what we will use in ggplot2 in order to create a density plot of the income of all customers in the ‘custdata’ dataset.

```
library(ggplot2)
ggplot(custdata, aes(x=income)) +
  geom_density(stat = "density", position = "identity") +
  coord_cartesian() +
  scale_x_continuous() +
  scale_y_continuous()
```



You can see that firstly, we have 'custdata' as **Data** in the *ggplot()* function.

Then, we have a data mapping specifying that income value will be mapped to the x-axis in **Aes**.

The **Geom** 'geom_density' indicates that we are using a density plot.

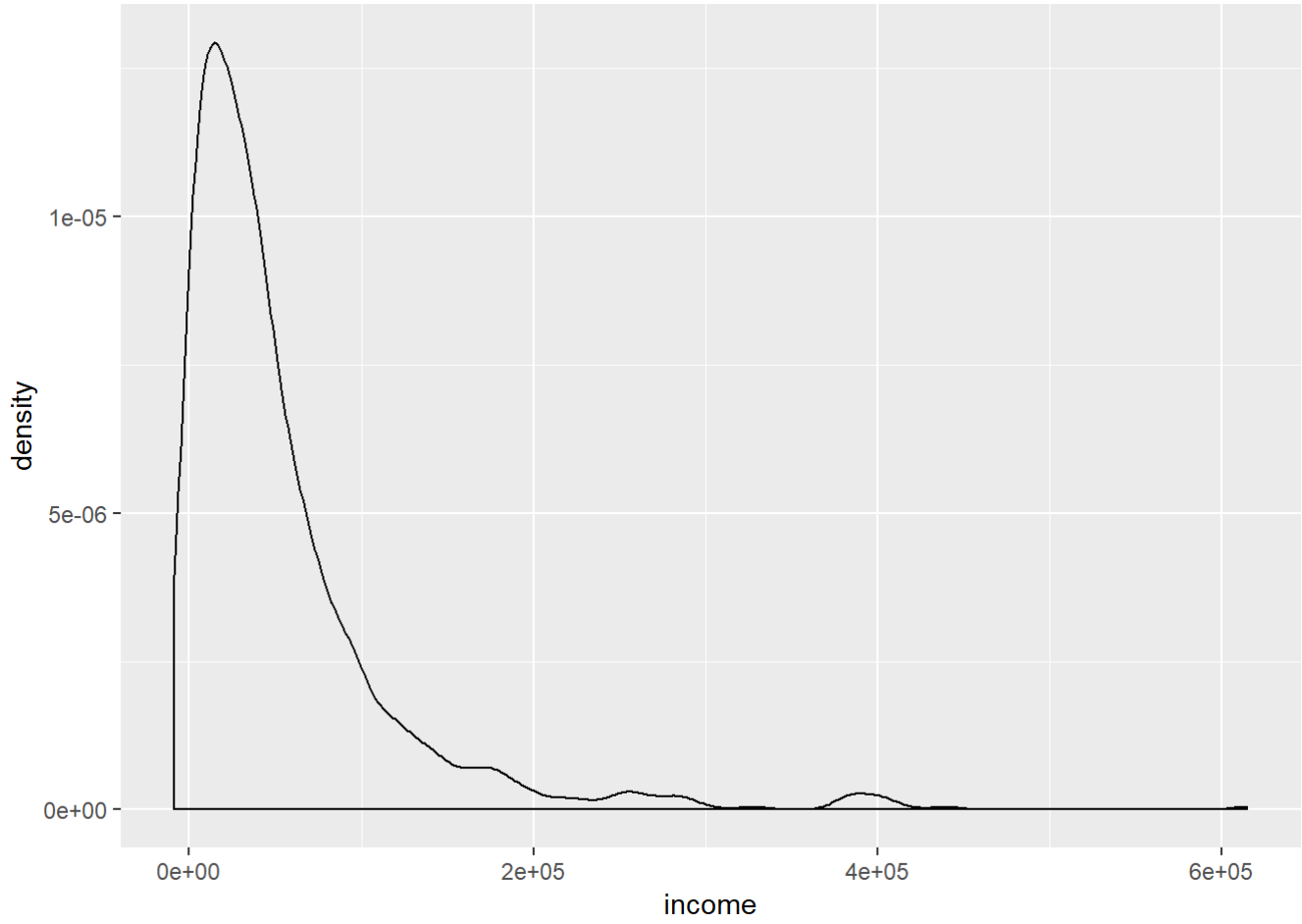
The **Stat** applied to the data for this plot is 'density'.

The **Position** of the geometric object is 'identity'.

We adopted the Cartesian **Coord** system and both the x and y-axis use continuous **Scale**.

However, this is just an example to show the various grammar of graphics for ggplot. In reality, many of these options are already default setting for density plots. A much more simplified command can achieve the same result.

```
ggplot(custdata, aes(x=income)) +
  geom_density()
```



Position

Here we will focus on the **Position** parameter in ggplot2.

There are *five types* of positioning:

Identity - default of most geoms.

Jitter - default of geom_jitter.

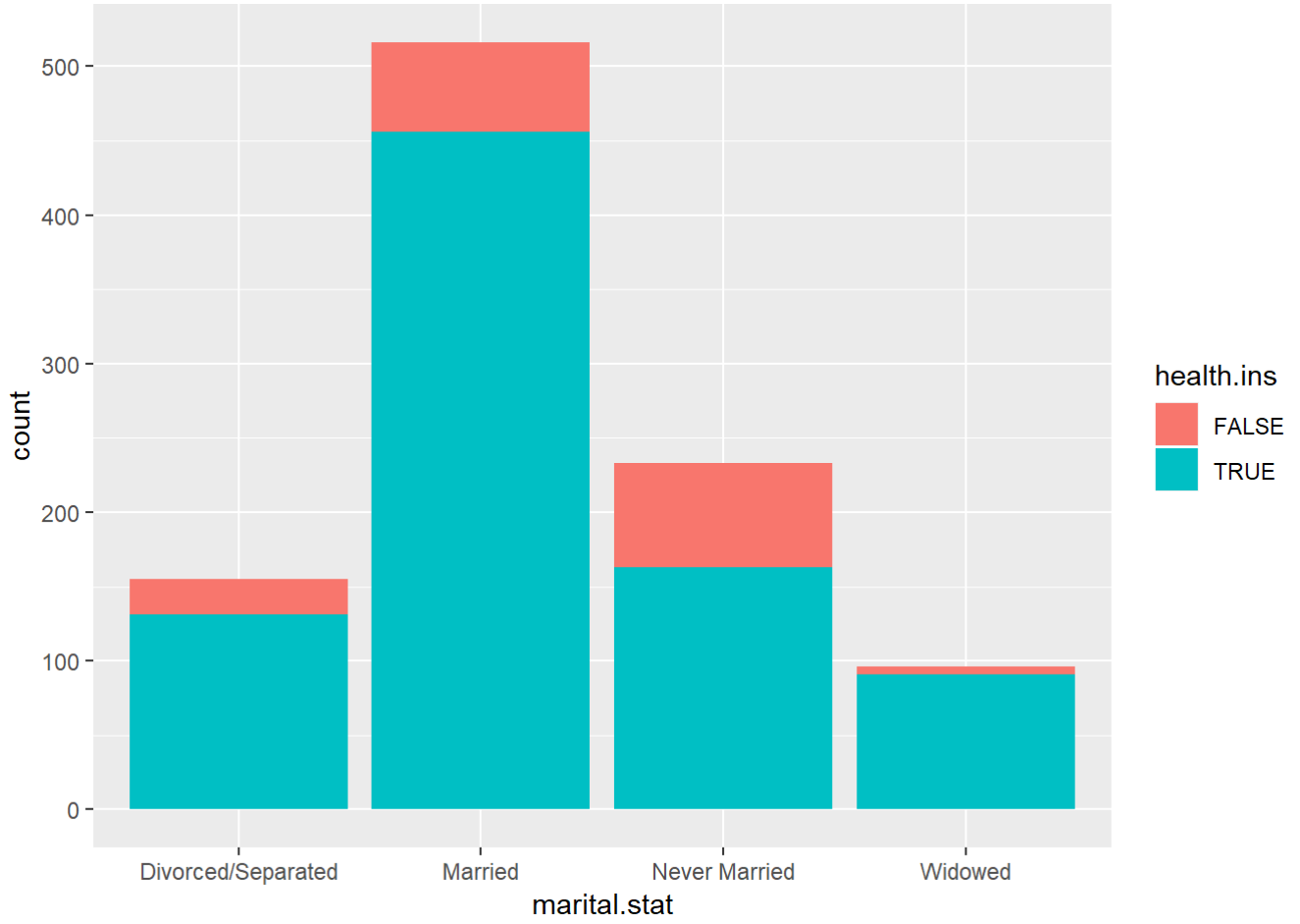
Dodge - default of geom_boxplot.

Stack - default of geom_bar, geom_histogram, and geom_area.

Fill - useful for geom_bar, geom_histogram, and geom_area.

Looking back at our 'custdata' dataset, how would we compare the number of people buy health insurance across different marital status? We would make use of 'geom_bar'.

```
ggplot(custdata) + geom_bar(aes(x=marital.stat, fill=health.ins))
```

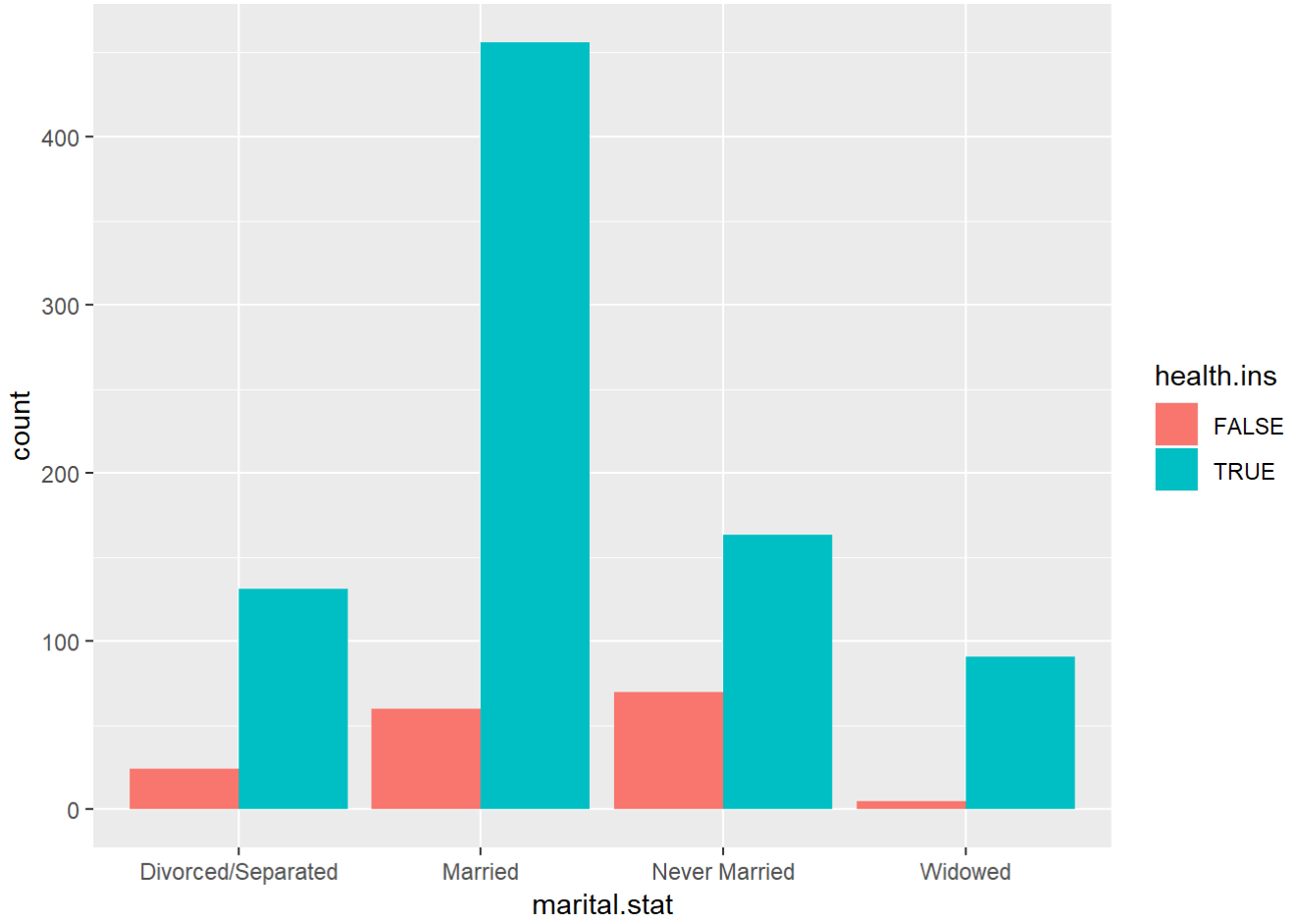


As **Stack** is the default **Position** parameter for geom_bar, the bars of different health insurances statuses are stacked vertically atop one another.

Here we see that 'Married' customers comprise of the largest segment those insured. 'Widowed' customers are the smallest segment that isn't insured.

Sometimes, comparing numbers across **Stack** positioning might be difficult. Instead, we can use the **Dodge** positioning to put bars side by side.

```
ggplot(custdata) +
  geom_bar(aes(x=marital.stat, fill=health.ins), position='dodge')
```

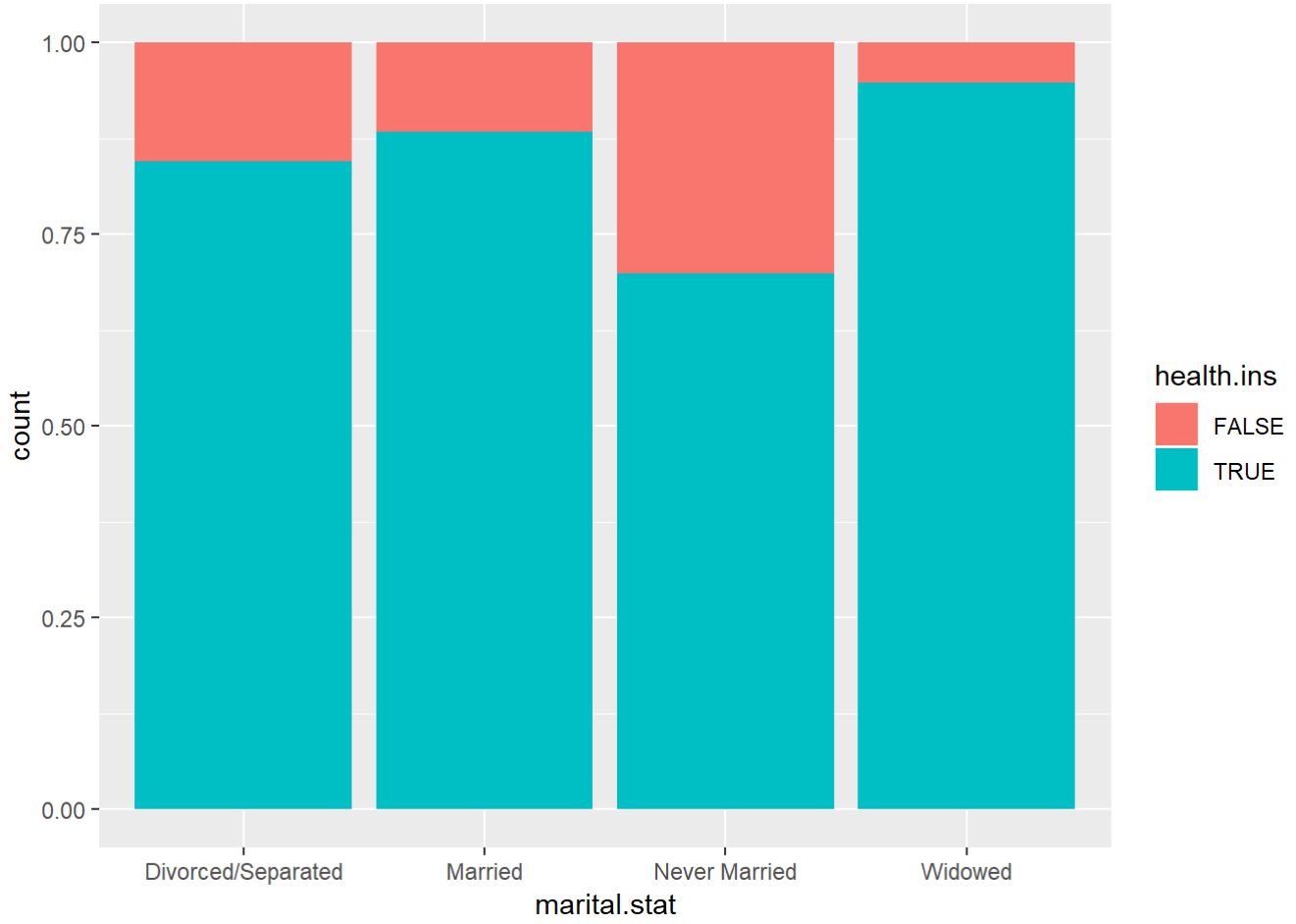


This allows us to compare numbers in-category and also across different categories.

Can we then conclude that because 'Married' customers have the highest absolute numbers, they are the best customers for the insurance company due to their large segment size? It may just be that most customers tend to be married, and the absolute numbers may, in fact, be misleading

Using **Fill** we can get a clear view of relative proportions in each category.

```
ggplot(custdata) +
  geom_bar(aes(x=marital.stat, fill=health.ins), position='fill')
```



From the plot, we can see that 'Widowed' customers clearly have the highest tendency of buying insurance, rather than those that are 'Married'. 'Never Married' customers also have the least tendency to buy health insurance by far.

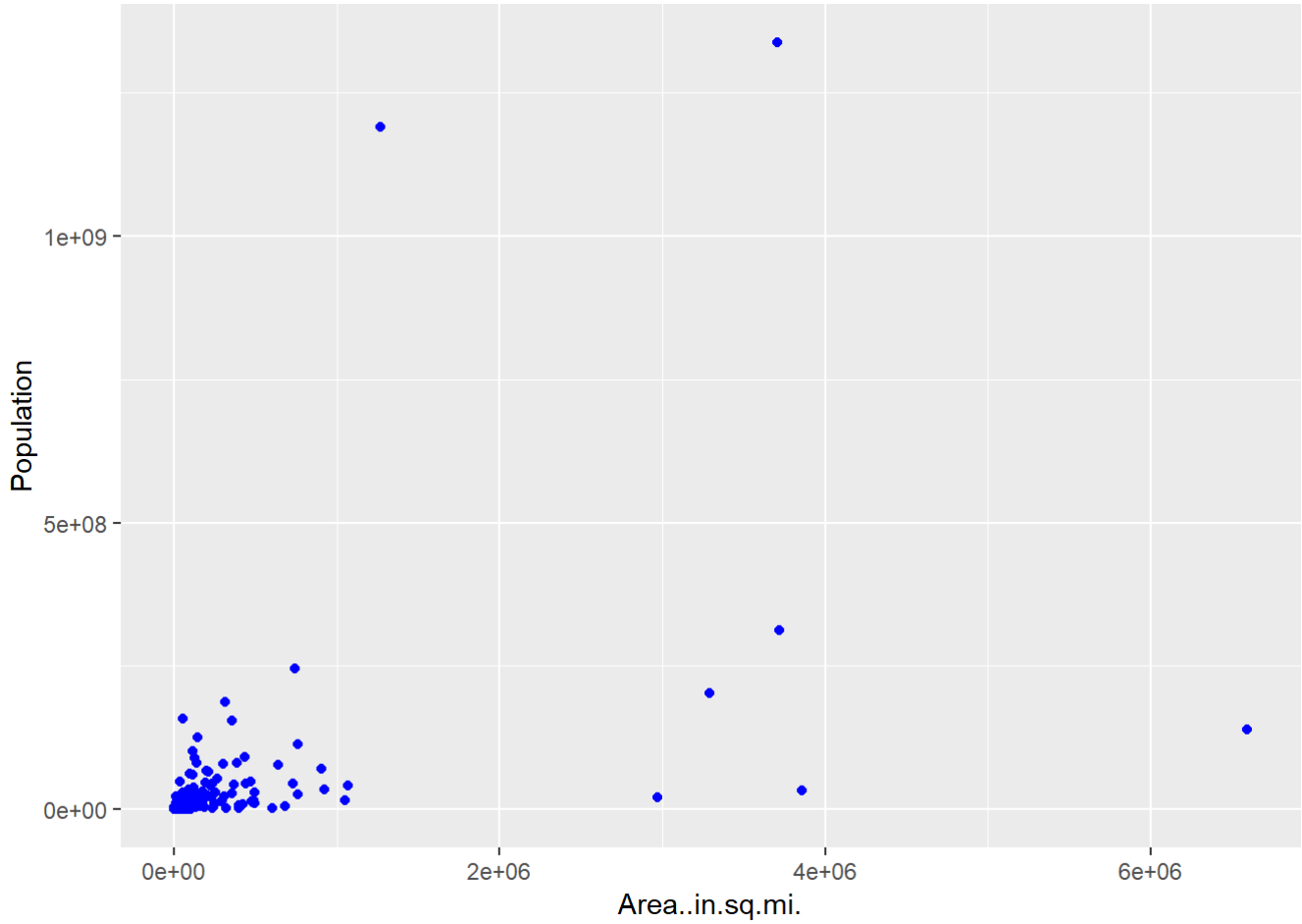
Data Transformation

We transform data for modeling needs, better visualization or better interpretation.

```
data <- read.csv("area&population-raw.csv")
```

Here, we show a scatter plot of the area against the population for all countries.

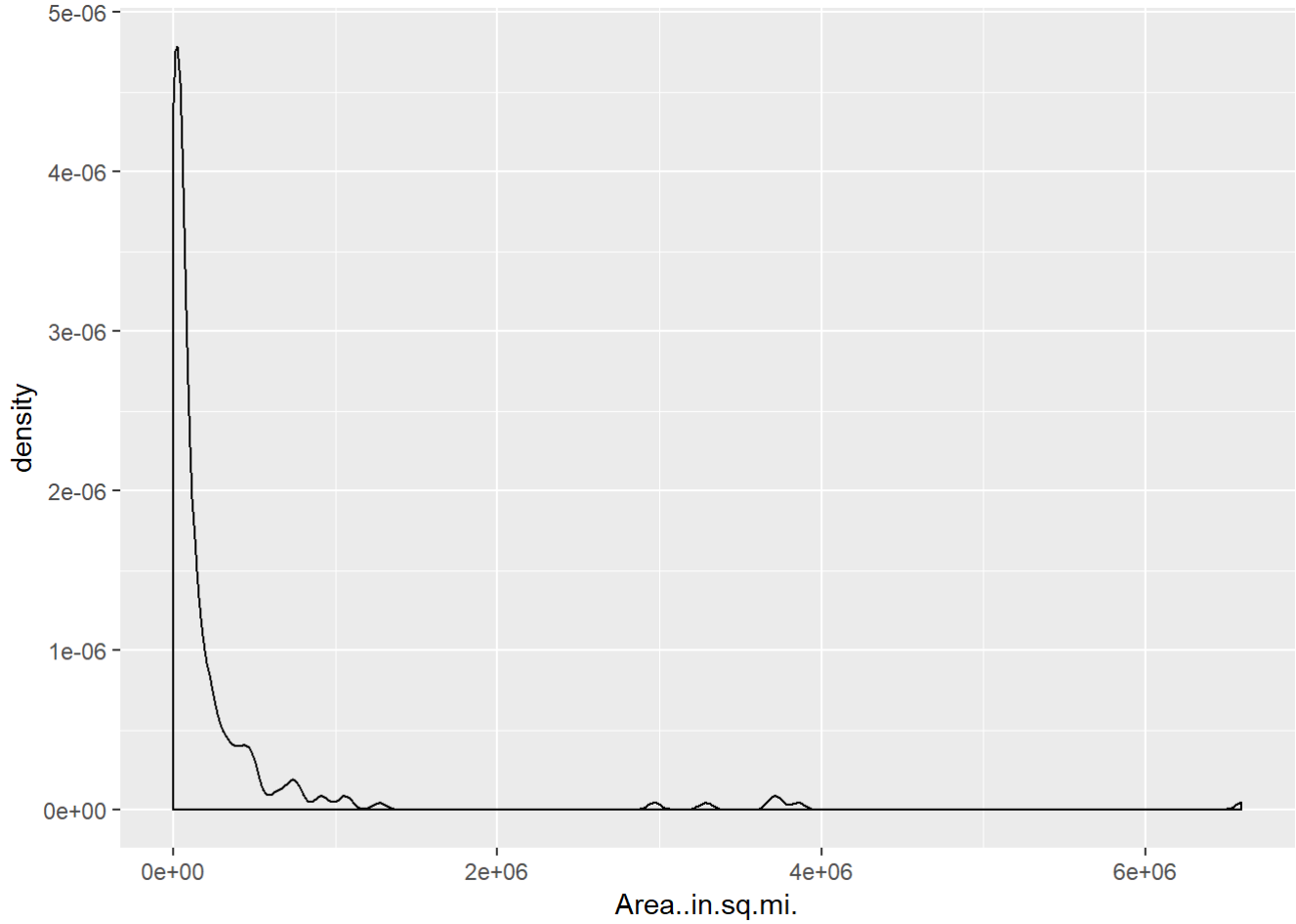
```
ggplot(data, aes(x=Area..in.sq.mi., y=Population)) +
  geom_point(colour='blue')
```



There is no discernable relationship between Population and Area in this plot. This is a poor visualization as most points are in the bottom left corner, except for some outliers.

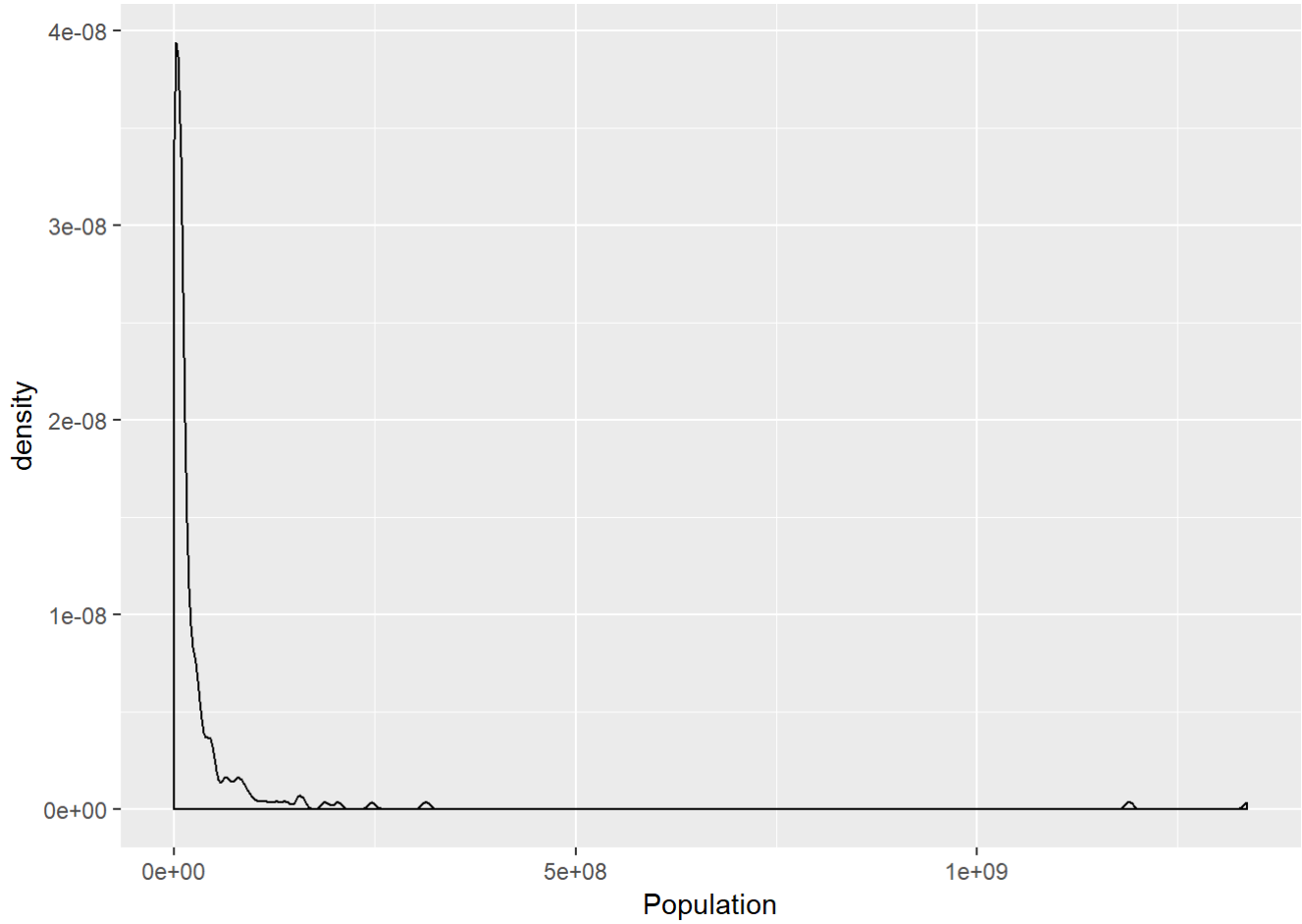
Let's develop a better understanding of our data, maybe that will shed some light on how to improve our plot. First, we can plot a density plot of the area.

```
ggplot(data, aes(x=Area..in.sq.mi.)) +
  geom_density()
```



The graph seems to have a normal distribution, but with a significant right skew. It is, in fact, a typical log-normal distribution. Which means its logarithm might follow a normal distribution.

```
ggplot(data, aes(x=Population)) +
  geom_density()
```

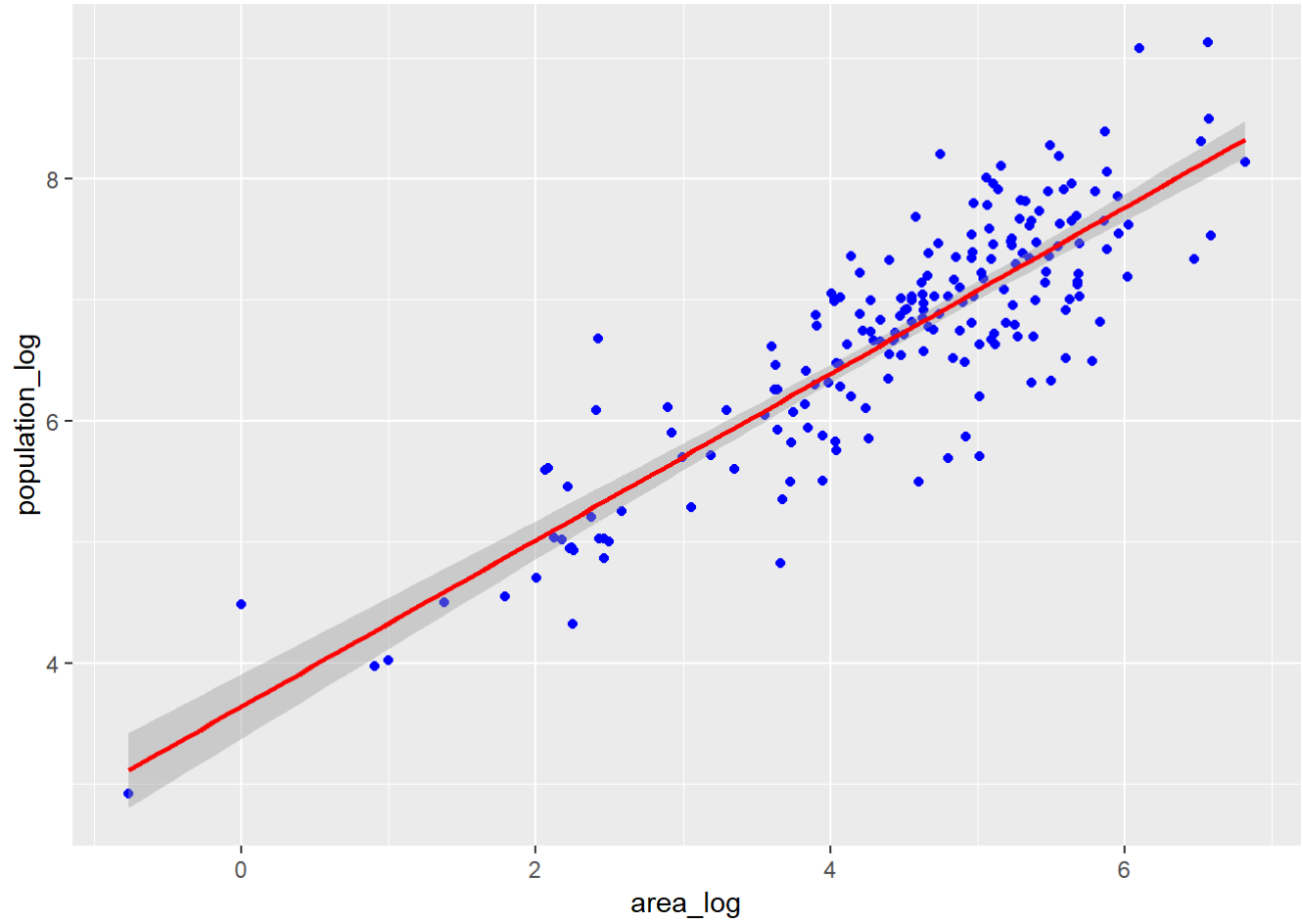


Notice that the population is also log-normally distributed. We should do a logarithm transformation in both area and population.

```
data$area_log = log10(data$Area..in.sq.mi.)
data$population_log = log10(data$Population)
```

A scatter plot of ‘area_log’ against ‘population_log’.

```
ggplot(data, aes(x=area_log, y=population_log)) +
  geom_point(colour='blue') +
  geom_smooth(method = 'lm', color='red')
```



Notice how every point is shown clearly, and it also shows a clear linear relationship between 'area_log' and 'population_log'. A simple linear regression could build a strong linear model between the two variables. It is also easy for us to interpret the relationship between area and population as well.

Suppose that the linear relationship between 'population_log' and 'area_log' is:

> population_log = a x area_log + c

For country 1, we have:

> Country 1: population_log1 = a x area_log1 + c

Suppose that:

> Country 2: area2 = n x area1

If we log this equation:

```
area_log2 = area_log1 + log(n)
```

With our first linear equation we can derive 'population_log2':

```
population_log2 = a x area_log2 + c
                = a x(area_log1 + log(n)) + c
                = a x area_log1 + alog(n) + c
                = a x area_log1 + c + alog(n)
                = population_log1 + alog(n)
```

Using the exponential function, we have:

```
population2 = n x 10^a x population1
```

Note that 10^a is a constant, independent of n. Meaning that if the area of a country was to increase by n times, the population will increase proportionally by a factor of 10^a x n as well.

Breaking Numerical Data Into Categories

Sometimes, we need to break numerical values into categories and study data behavior in these categories. For example, we might classify people by income group, classify products by sales volume or classify days by rainfall amount, etc.

Often, numerical values are broken down into intervals of equal length or divided by convenient numbers. However, it can be risky. It might not be fair to classify people by 0-2 years old vs 20-22 years old. Babies' behavior changes significantly from year to year, while youth's behavior doesn't change so dramatically across the same period.

It may instead be more reasonable to break down the age group of babies further into 0-1 month, 1-3 months, 3-6 months, etc.

Breaking numerical values into convenient numbers don't have much statistical foundation, except that the numbers look nice. We may break down annual incomes into 0-10,000, 10,000-50,000, etc. The difference between people earning 9,999 and 10,001 may be insignificant. Instead, data visualization may be a better way to classify numerical values.

```
ggplot(custdata, aes(x=income, y=as.numeric(health.ins))) +
  geom_point(position=position_jitter(w=0.05, h=0.05)) +
  scale_x_log10(breaks=c(100,1000,10000,100000)) +
  geom_smooth() +
  annotation_logticks(sides="bt")
```

```
## Warning in self$trans$transform(x): NaNs produced
```

```
## Warning: Transformation introduced infinite values in continuous x-axis
```

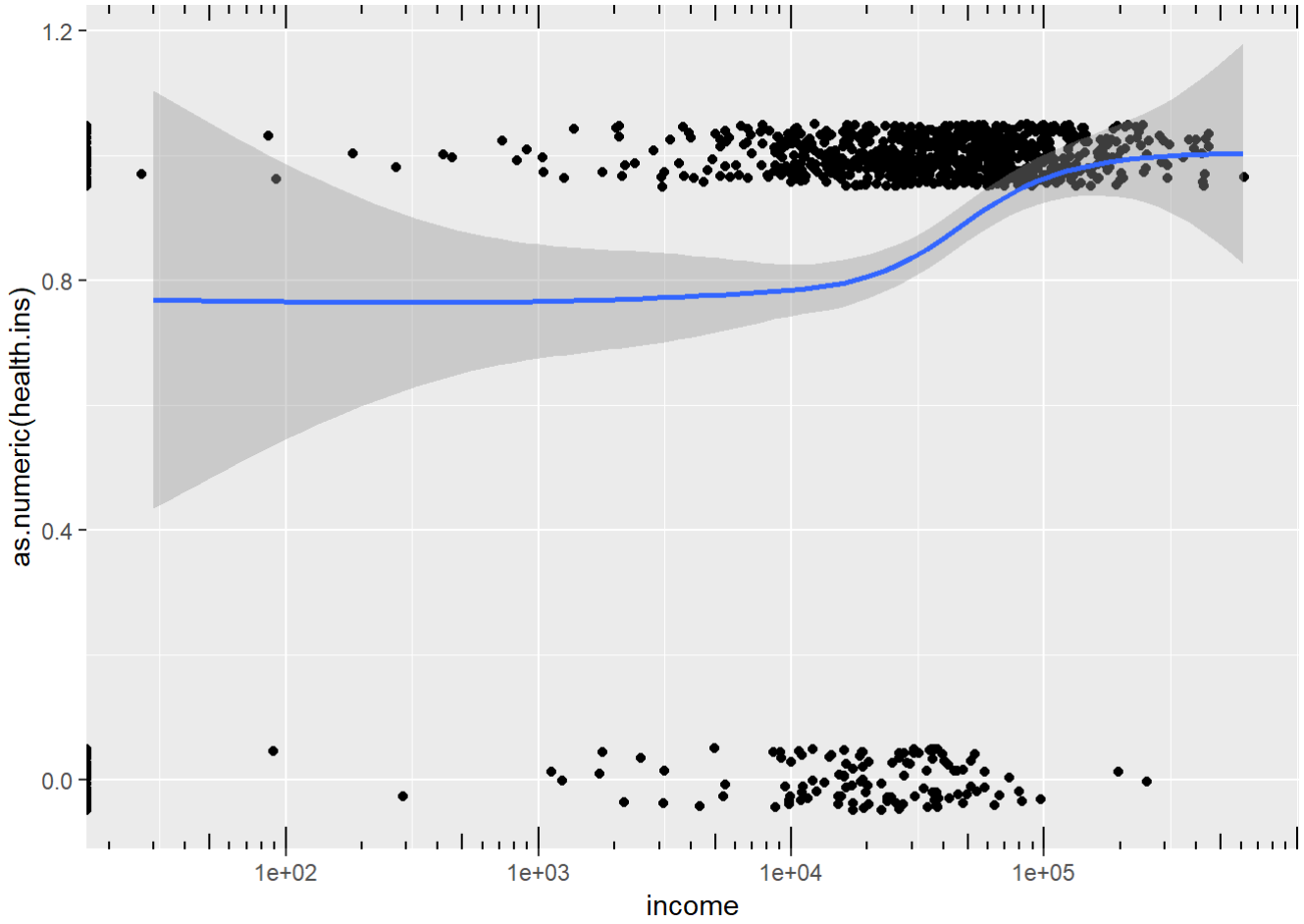
```
## Warning in self$trans$transform(x): NaNs produced
```

```
## Warning: Transformation introduced infinite values in continuous x-axis
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

```
## Warning: Removed 79 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 1 rows containing missing values (geom_point).
```



The plot shows customer income vs whether they buy health insurance. The blue smoothing curve in the middle of the figure is the best fit curve approximating the ratio of customers buying health insurance with respect to their income.

Notice that the curve is quite flat when income is below 20,000, but increases sharply after income exceeds 20,000 and becomes flatter again after income exceeds 100,000. This gives us a hint that we should break income into three groups, 0-20,000, 20,000-100,000, and >100,000 if we are interested in how income affects the decision to buy health insurance.

It is important to understand your data before taking action. The foundation of what descriptive analytics is about.

Data Normalisation

Absolute figures compared across two categories might not make much sense before normalization. As having SGD$2,000 income in Singapore affords a vastly different lifestyle from SGD$2,000 income in Indonesia. It is not just the exchange rate and the cost of living, but also the difference in wealth between the two nations that make such a direct comparison tricky.

We can first normalize the data in order to compare incomes across the two countries. Using the median or average income in each nation as the benchmark and generate a relative ratio between each person's income against the benchmark of the country they live in.

Looking back at our 'custdata' dataset, we can see that customers in the dataset come from different states and it would be unfair to directly compare their incomes. The income levels in different states vary a lot.

Here is the script from 'GetMedianIncome.R'.

```
#This script will obtain USA state medium incomes

library(XML)
library(RCurl)
```

```
## Loading required package: bitops
```

```
theurl<-"https://en.wikipedia.org/wiki/List_of_U.S._states_and_territories_by_income"
urldata <- getURL(theurl)
data<-readHTMLTable(urldata,stringAsFactors=FALSE)
median.income <- as.data.frame(data[4])
colnames(median.income) <- c("Rank","State","Per.Capita.Income","Median.Household.Income","Me
dian.Family.Income","Population","No.Households","No.Families")
median.income <- median.income[-c(1),]

#Convert Per.Capita.Income from text to numeric
median.income$Per.Capita.Income <- as.numeric(gsub(",","",gsub("\\$","",median.income$Per.Cap
ita.Income)))
```

We now have the dataset 'median.income' that contains per capita income across the different states of the US.

We will now merge this with our 'custdata' dataset.

```
custdata <- merge(custdata, median.income, by.x = "state.of.res",
                  by.y = "State")
```

We will then normalize the income with this command.

```
custdata$income.normalised <- with(custdata,
                                   income/Per.Capita.Income)
```

We can then use these normalized income values instead of the original income values.

Ggmap

Spatial data is now widely available. Studying patterns or behaviors on a geographical basis is becoming a common interest for business and data scientists.

Ggmap allows us to visualize spatial data on a map. Remember to follow the pdf file from week 1 to setup google and ggmap.

'ggmap' is a package developed to work on top of ggplot2 for visualizing spatial data. So it is compatible with ggplot2. Ggmap can be viewed as simply another layer in ggplot2. This is advantageous as you can overlay all other ggplot2 plots with ggmap plots.

To plot a map of a country, state or city, you can obtain the map just by specifying its respective name.

```
library(ggmap)
```

```
## Google's Terms of Service: https://cloud.google.com/maps-platform/terms/.
```

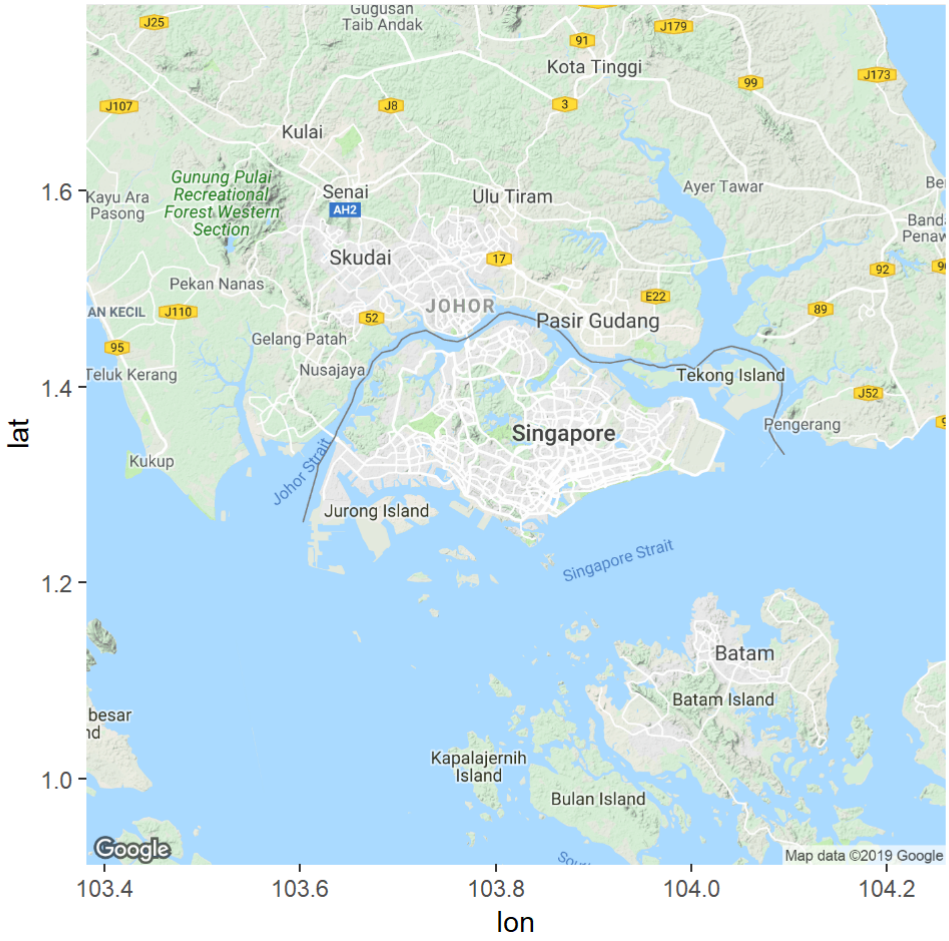
```
## Please cite ggmap if you use it! See citation("ggmap") for details.
```

```
m <- get_map("Singapore", source="google")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Singapore&zoom=10&size=640x640&scale=2&maptype=terrain&language=en-EN&key=xxx-fKwOanMfOTME
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Singapore&key=xxx-fKwOanMfOTME
```

```
ggmap(m)
```

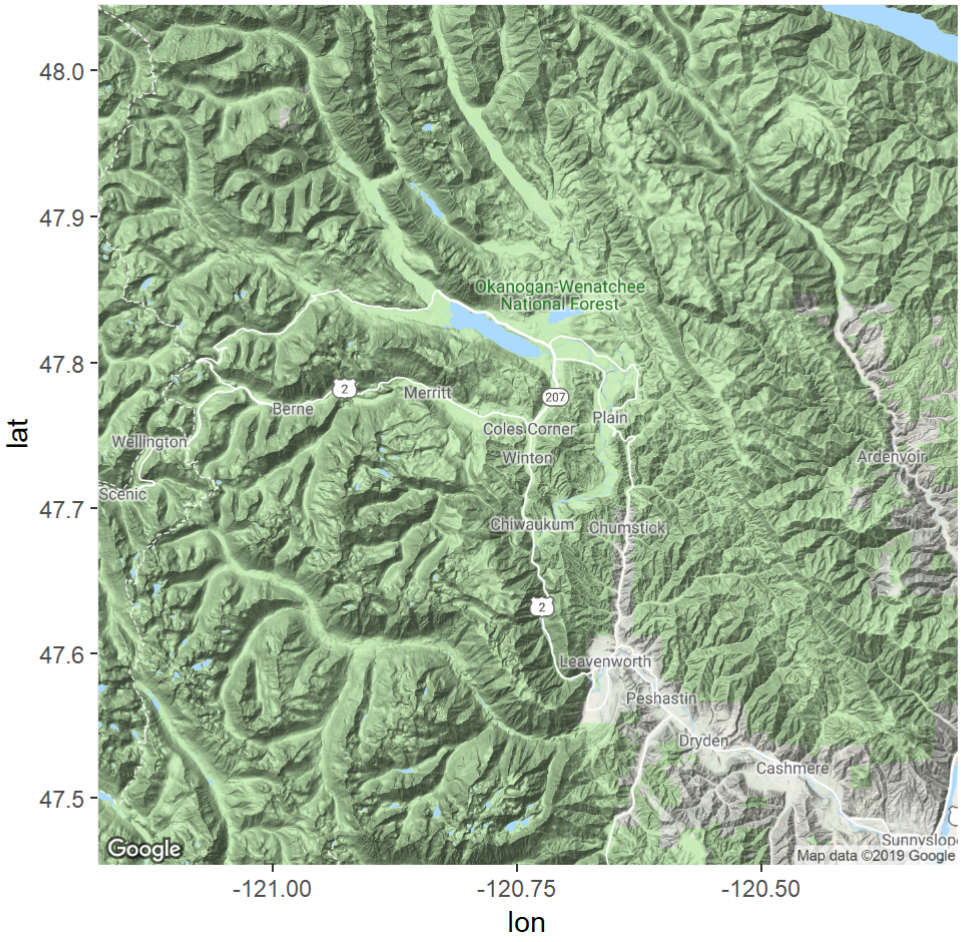



```
m <- get_map("Washington", source="google")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Washington&zoom=10&size=640x640&scale=2&maptype=terrain&language=en-EN&key=xxx-fKwOanMfOTME
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Washington&key=xxx-fKwOanMfOTME
```

```
ggmap(m)
```



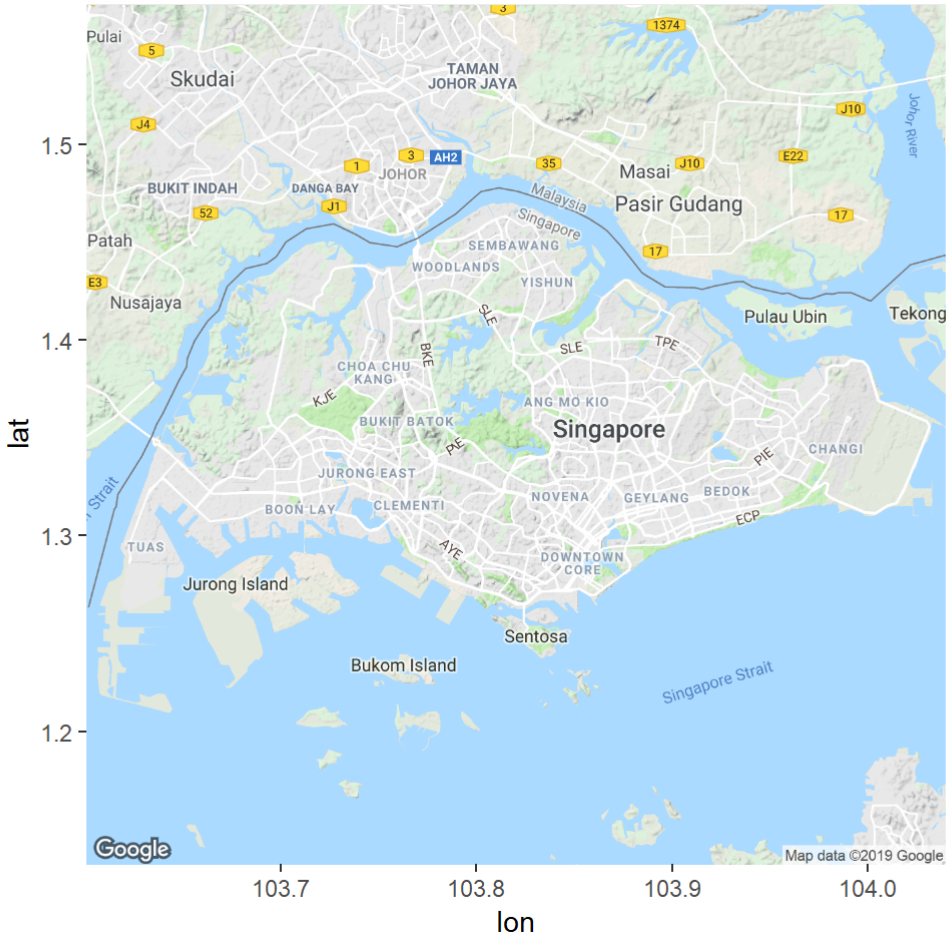
We can also use the zoom parameter to zoom in or out.

```
m <- get_map("Singapore", zoom=11, source="google")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Singapore&zoom=11&size=640x640&scale=2&maptype=terrain&language=en-EN&key=xxx-fKwOanMfOTME
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Singapore&key=xxx-fKwOanMfOTME
```

```
ggmap(m)
```



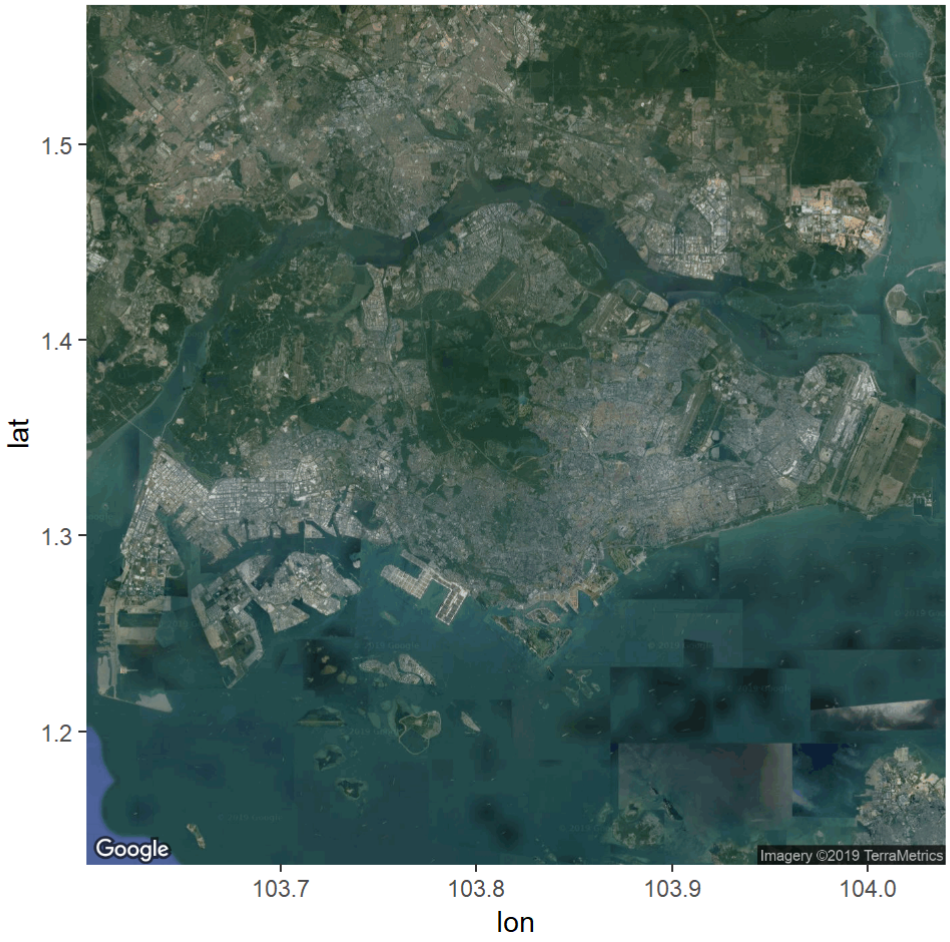
There are also several maptypes to choose from, such as *satellite*, *hybrid*, *toner*, *watercolor*, *terrain-background*, or *toner-lite*.

```
m <- get_map("Singapore", zoom=11, maptype="satellite",
             source="google")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Singapore&zoom=11&size=640x640&scale=2&maptype=satellite&language=en-EN&key=xxx-fKwOanMfOTME
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Singapore&key=xxx-fKwOanMfOTME
```

```
ggmap(m)
```



There are also four different sources of a map: *google*, *osm*, *stamen*, and *cloudmade*. Only stamen is open, with the rest requiring API keys.

```
m <- get_map("Singapore", zoom=11, source="stamen")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Singapore&zoom=11&size=640x640&scale=2&maptype=terrain&key=xxx-fKwOanMfOTME
```

```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Singapore&key=xxx-fKwOanMfOTME
```

```
## Source : http://tile.stamen.com/terrain/11/1613/1015.png
```

```
## Source : http://tile.stamen.com/terrain/11/1614/1015.png
```

```
## Source : http://tile.stamen.com/terrain/11/1615/1015.png
```

```
## Source : http://tile.stamen.com/terrain/11/1613/1016.png
```

```
## Source : http://tile.stamen.com/terrain/11/1614/1016.png
```

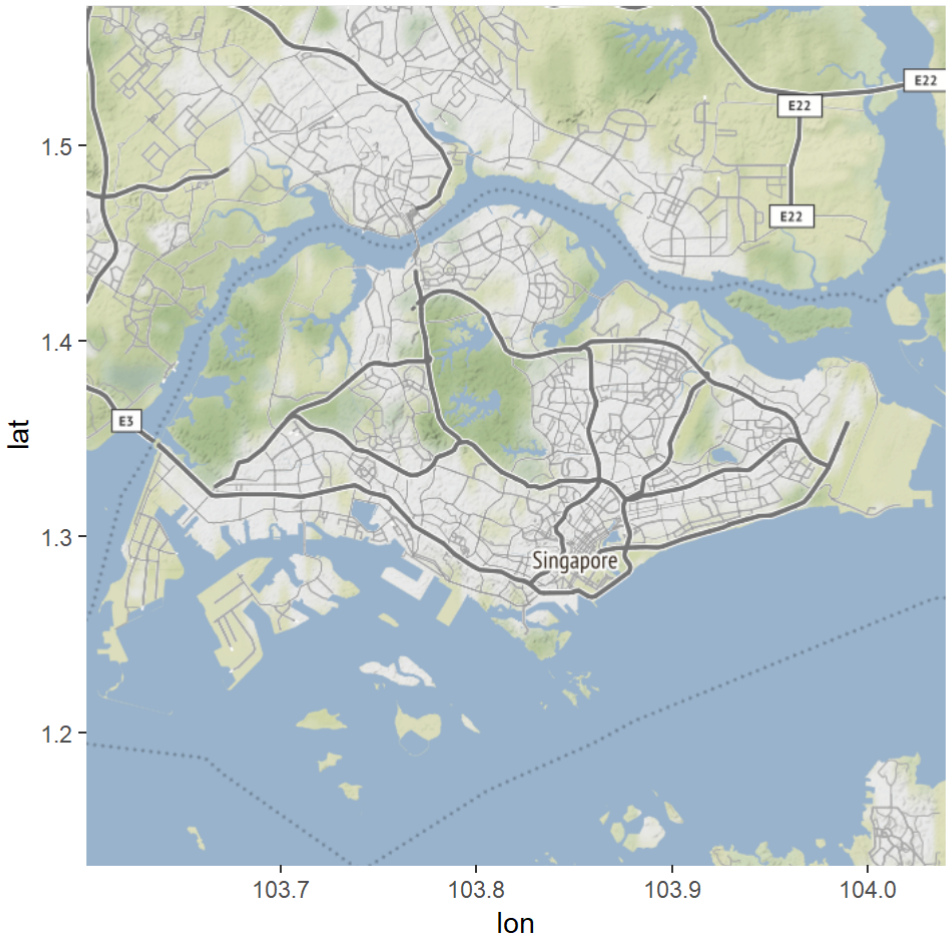
```
## Source : http://tile.stamen.com/terrain/11/1615/1016.png
```

```
## Source : http://tile.stamen.com/terrain/11/1613/1017.png
```

```
## Source : http://tile.stamen.com/terrain/11/1614/1017.png
```

```
## Source : http://tile.stamen.com/terrain/11/1615/1017.png
```

```
ggmap(m)
```



You can see that stamen's maps are not as nice as Google's. However, it is a possible alternative if you don't want to go through the trouble of enabling Google API key.

Case Study: Pizza Hut

Here we will use Pizza Hut location data to illustrate how to integrate ggmap with other ggplot2 functions.

```
pizzahut.location <- read.csv("PizzaHut.csv", header = TRUE,
                              colClasses = c("character",
                                             "character",
                                             "factor",
                                             "character",
                                             "numeric",
                                             "numeric"))
```

This data contains the address, region, longitude and latitude of all Pizza Hut branches in Singapore.

```
map <- get_map("Singapore", zoom=11, source="google")
```

```
## Source : https://maps.googleapis.com/maps/api/staticmap?center=Singapore&zoom=11&size=640x640&scale=2&maptype=terrain&language=en-EN&key=xxx-fKwOanMfOTME
```

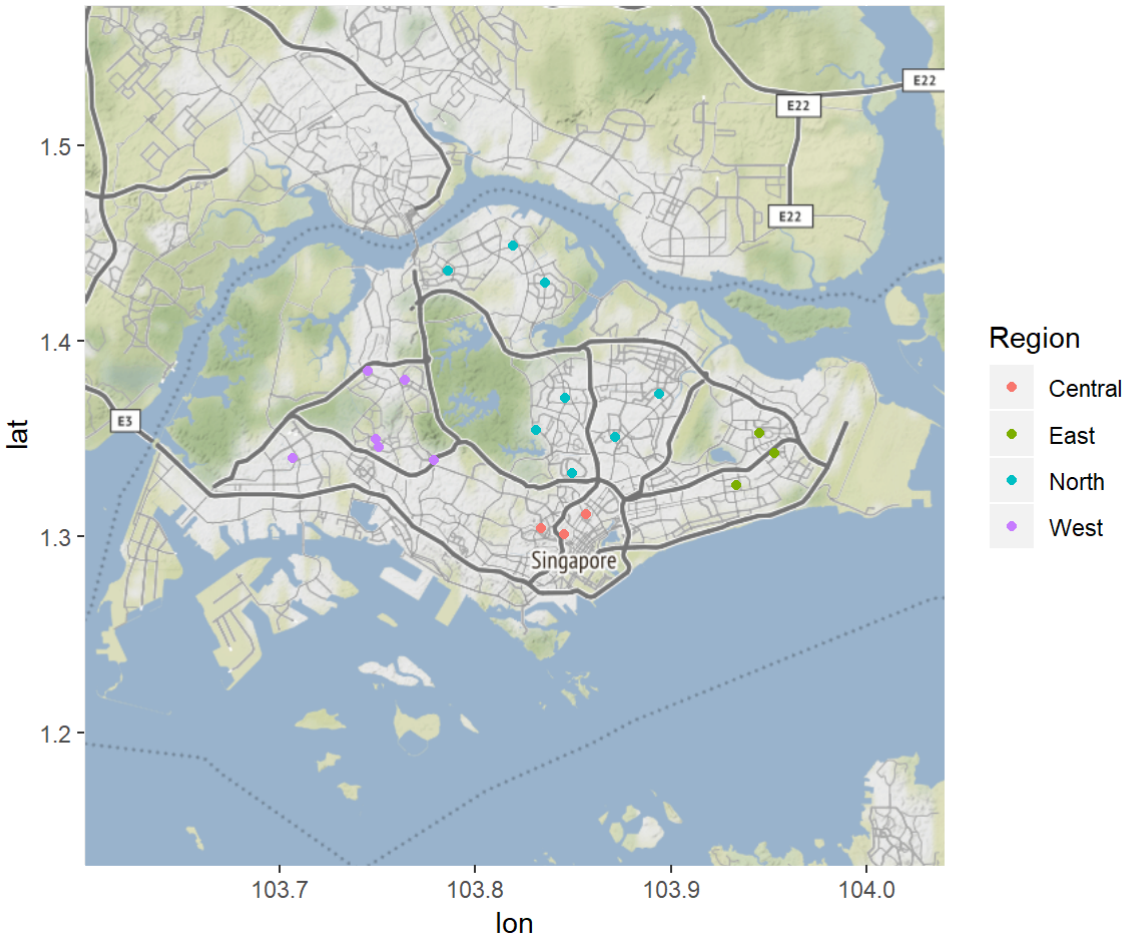
```
## Source : https://maps.googleapis.com/maps/api/geocode/json?address=Singapore&key=xxx-fKwOanMfOTME
```

Next, get the map and then add ggplot as the base layer to the map.

```
m1 <- ggmap(m, base_layer = ggplot(data = pizzahut.location,
                                   aes(x=lon, y=lat)))
```

Finally, use a “geom_point” function to plot points on the location of pizza hut branches, given by the longitude, which is mapped to the x-axis and the latitude, which is mapped to the y-axis in the plot. The points are further colored by the region they belong to.

```
m1 + geom_point(aes(color=Region))
```



You are also highly recommended to read how to integrate Heatmap with ggmap (https://blog.dominodatalab.com/geographic-visualization-with-rs-ggmaps/)

Map View by Borders

Sometimes we may want to compare data by regions, not by specific locations. “ggmap” package is actually insufficient for this. Instead, we will have to use the ‘raster’ and ‘rgdal’ packages.

Use the code below to install them, or use the packages pane in Rstudio.

```
install.packages("raster")
install.packages("rgdal")
```

The key function is *getData()* from 'raster', which will obtain geographic data for anywhere in the world. It currently supports five sources of data. *GADM*, *countries*, *SRTM*, *alt*, and *world-clim*.

```
library(rgdal)
```

```
## Loading required package: sp
```

```
## rgdal: version: 1.3-9, (SVN revision 794)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: C:/Users/Alex/Documents/R/win-library/3.5/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/Users/Alex/Documents/R/win-library/3.5/rgdal/proj
##  Linking to sp version: 1.3-1
```

```
library(raster)
library(XML)
SG <- getData('GADM', country='SG', level =1)
SG
```

```
## class       : SpatialPolygonsDataFrame
## features    : 5
## extent      : 103.6091, 104.0858, 1.16639, 1.471388  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 10
## names       : GID_0,    NAME_0,   GID_1,  NAME_1, VARNAME_1, NL_NAME_1, TYPE_1, ENGTYPE_1,
CC_1, HASC_1
## min values  :   SGP, Singapore, SGP.1_1, Central,        NA,        NA, Region,    Region,
NA,     NA
## max values  :   SGP, Singapore, SGP.5_1,    West,        NA,        NA, Region,    Region,
NA,     NA
```

This will get us Singapore map data from GADM. Its class is "SpatialPolygonsDataFrame", which is a special class wrapped around the data frame. Unlike a normal data frame, we will use at '@' instead of dollar sign '$'.

The most important variables are 'data' and 'polygons'.

```
SG@data
```

```
##        GID_0    NAME_0   GID_1     NAME_1 VARNAME_1 NL_NAME_1 TYPE_1
## 260003   SGP Singapore SGP.1_1    Central      <NA>      <NA> Region
## 260001   SGP Singapore SGP.2_1       East      <NA>      <NA> Region
## 259999   SGP Singapore SGP.3_1      North      <NA>      <NA> Region
## 260000   SGP Singapore SGP.4_1 North-East      <NA>      <NA> Region
## 260002   SGP Singapore SGP.5_1       West      <NA>      <NA> Region
##        ENGTYPE_1 CC_1 HASC_1
## 260003    Region <NA>   <NA>
## 260001    Region <NA>   <NA>
## 259999    Region <NA>   <NA>
## 260000    Region <NA>   <NA>
## 260002    Region <NA>   <NA>
```

```
class(SG@data)
```

```
## [1] "data.frame"
```

This shows us that ‘SG@data (mailto:SG@data)’ itself is a data frame. ‘NAME_0’ stores the name of the country while ‘NAME_1’ stores the names of the five regions in Singapore.

GADM only has data up to this level for Singapore. However, for bigger countries, GADM will have data up to _2, where 1 are the states and 2 are the counties.

SG@Data (mailto:SG@Data) is the outer layer for the SG object to communicate with other data sets. For example, suppose we create a data frame storing some numbers for each region in Singapore.

```
data_1 <- data.frame(Region=c('Central', 'East', 'North',
                              'North-East', 'West'),
                     value=c(3,4,1,7,10))
data_1
```

```
##       Region value
## 1    Central     3
## 2       East     4
## 3      North     1
## 4 North-East     7
## 5       West    10
```

Then we want to merge SG@data (mailto:SG@data) with data_1.

```
SG@data <- merge(SG@data, data_1, by.x='NAME_1', by.y='Region')
SG@data
```

```
##       NAME_1 GID_0    NAME_0   GID_1 VARNAME_1 NL_NAME_1 TYPE_1 ENGTYPE_1
## 1    Central   SGP Singapore SGP.1_1      <NA>      <NA> Region    Region
## 2       East   SGP Singapore SGP.2_1      <NA>      <NA> Region    Region
## 3      North   SGP Singapore SGP.3_1      <NA>      <NA> Region    Region
## 4 North-East   SGP Singapore SGP.4_1      <NA>      <NA> Region    Region
## 5       West   SGP Singapore SGP.5_1      <NA>      <NA> Region    Region
##   CC_1 HASC_1 value
## 1 <NA>   <NA>     3
## 2 <NA>   <NA>     4
## 3 <NA>   <NA>     1
## 4 <NA>   <NA>     7
## 5 <NA>   <NA>    10
```

The values from data_1 are now added to SG@data (mailto:SG@data) by its respective regions.

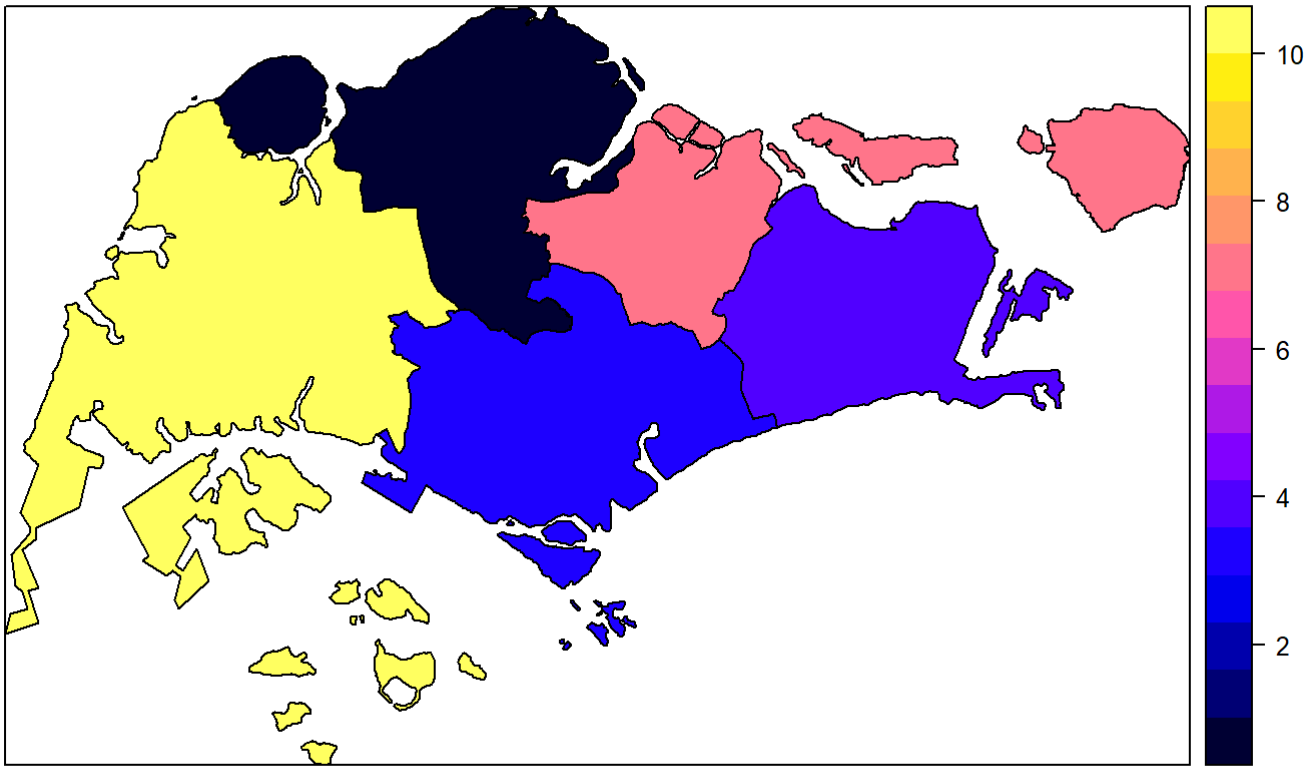
Each region in Singapore will later be plotted as a polygon based on data stored in SG@Polygons (mailto:SG@Polygons).

```
class(SG@polygons)
SG@polygons[1]
```

It shows a list of location data which is corresponding to the corners of polygons of region 1. (in order to not needlessly lengthen this file, I will not show the element here. However, you can copy and run this yourself)

This is just for your information, and will never touch this data.

```
spplot(SG, 'value')
```



It will plot the Singapore map with the given polygon data and color the five regions using different colors based on the values stored in the 'value' column of SG@data (mailto:SG@data) which originally comes from "data_1".

In summary, to generate map view by borders, first, download map data from the right source. Next, link your data to the map data. Lastly, plot the map.

Leaflet

A great alternative to ggmap is leaflet. JavaScript is THE programming for the web. Leaflet is one of the most popular open-source JavaScript libraries for interactive maps.

Leaflet package in R is the package for you to integrate and control the JavaScript leaflet library. With a few lines of code, you can make use of the powerful features of the leaflet library.

Refer back to the original work (https://rpubs.com/VMS/335299) if needed.

Say that we want to show Sentosa Island on a map. Think of an address on the island, such as Sentosa Cove. Visit this website (https://www.latlong.net/convert-address-to-lat-long.html) and type in "Sentosa Cove Singapore". Click "Find" and you will get latitude of 1.239660, longitude of 103.835381.

First, we need to create a leaflet widget.

```
library(leaflet)
library(tidyverse)
```

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

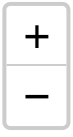
```
## v tibble  2.0.1     v purrr   0.2.5
## v tidyr   0.8.2     v dplyr   0.7.8
## v readr   1.2.1     v stringr 1.3.1
## v tibble  2.0.1     v forcats 0.3.0
```

```
## -- Conflicts ------------------------------------ tidyverse_conflicts() --
## x tidyr::complete() masks RCurl::complete()
## x tidyr::extract()  masks raster::extract()
## x dplyr::filter()   masks stats::filter()
## x dplyr::lag()      masks stats::lag()
## x dplyr::select()   masks raster::select()
```

```
m <- leaflet()
```

Then, we add a specific map layer to it.

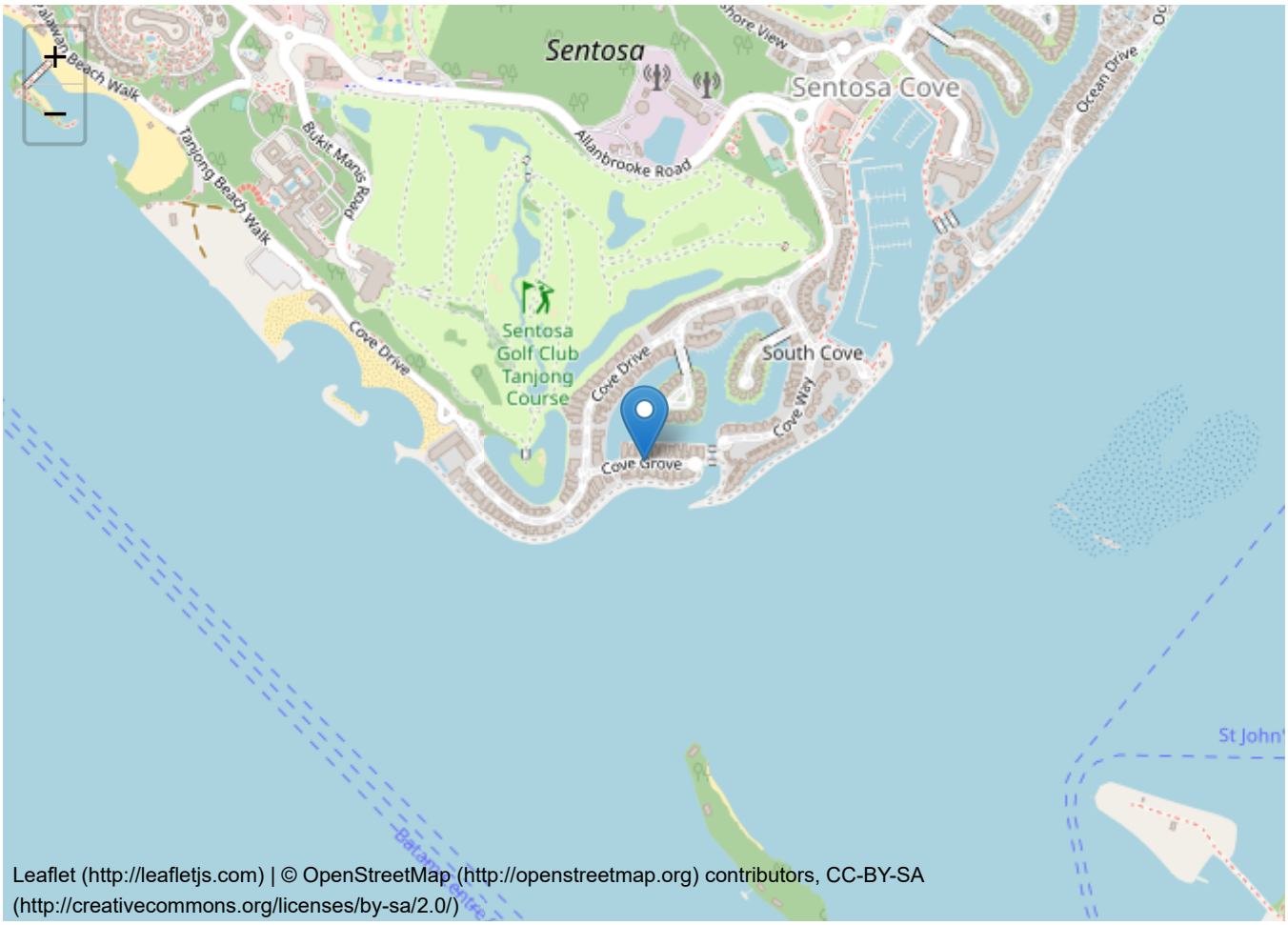
```
m <- addTiles(m)
m
```



It is basically an empty world map, waiting for a specific instruction from us. Leaflet provides many functions for us to add all sorts of layers on top of the base layer.

One of the most popular functions is addMarkers.

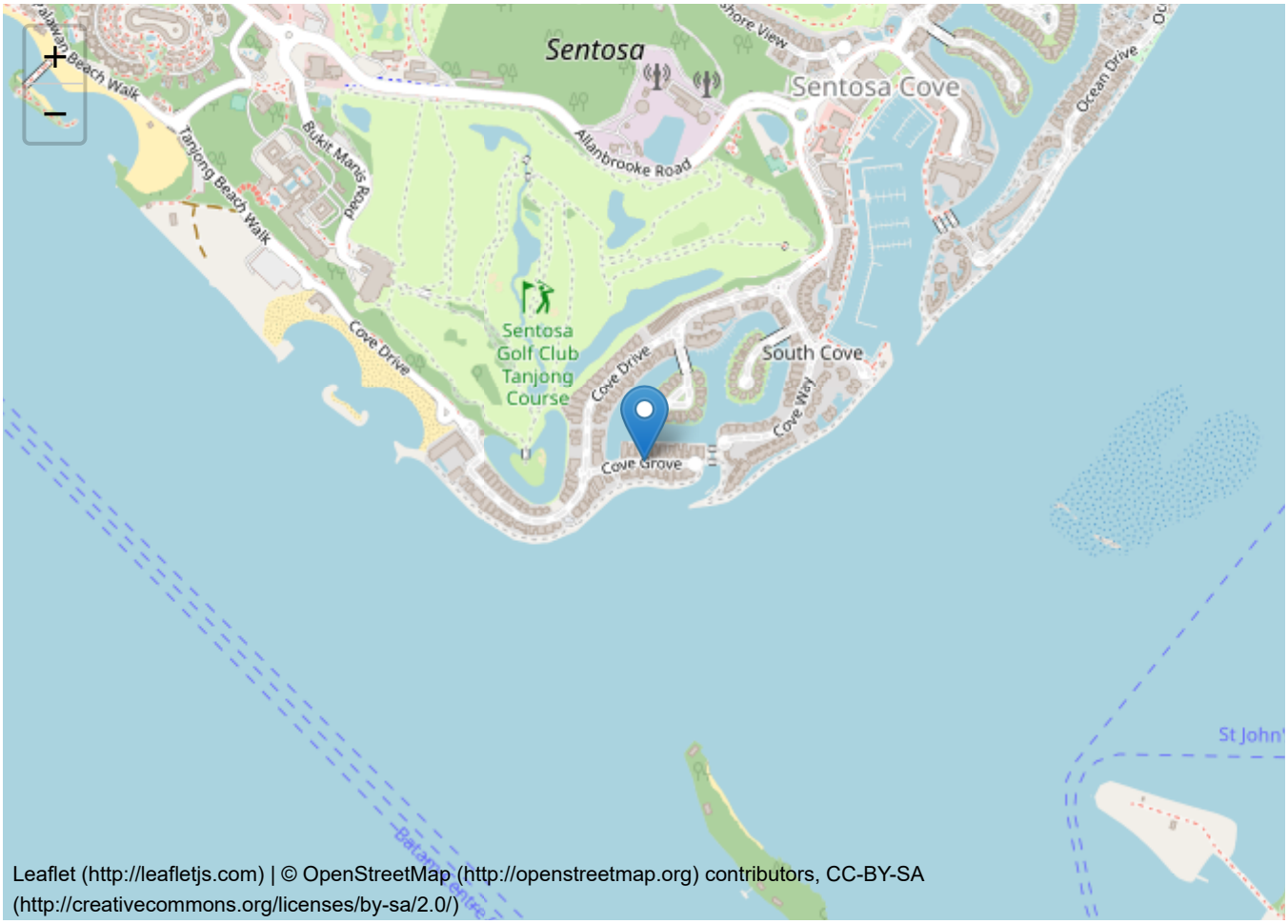
```
m <-addMarkers(m, lat = 1.239660, lng = 103.835381,
               popup = "Sentosa Cove")
m
```



This produces a map that centers at Sentosa Cove. The map is interactive, by clicking on the marker, the popup message we set in the function will pop up. We can also zoom in or zoom out on the map.

The above steps can be simplified into one line of code.

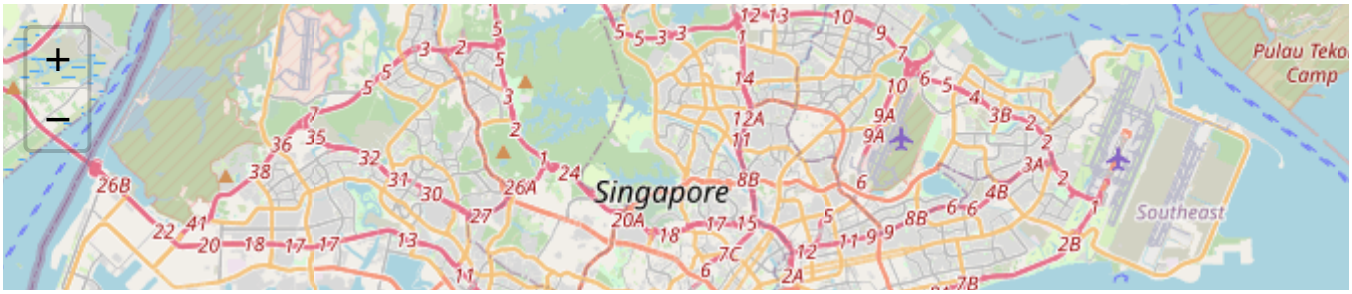
```
m <- leaflet() %>% addTiles() %>%
  addMarkers(lat = 1.239660, lng = 103.835381,
             popup = "Sentosa Cove")
m
```

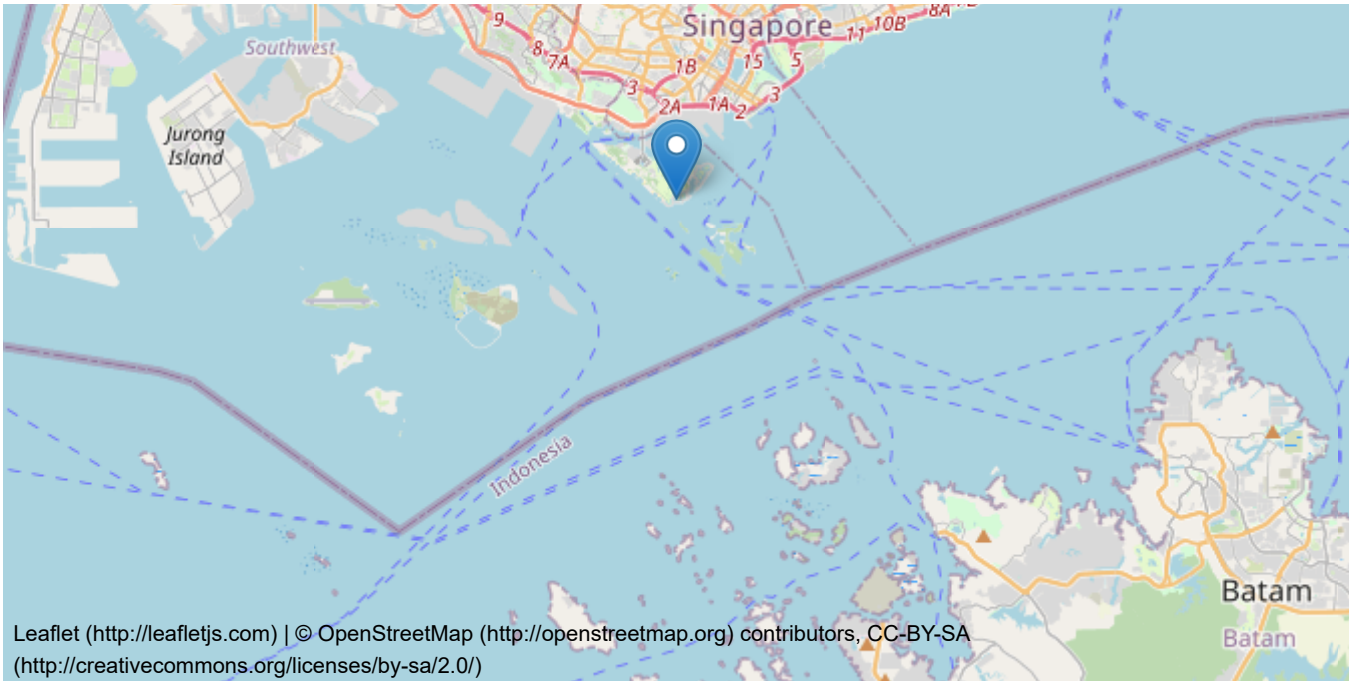


If our objective is to show tourist attractions in Singapore with Sentosa Island as just one of the many spots, then what we have now is not ideal.

Instead, we can pre-set the zoom level so as to show the whole of Singapore by default. This can be done by adding one more function, SetView in the code.

```
m <- leaflet() %>% setView(lat = 1.239660, lng = 103.835381,
                           zoom = 11) %>%
  addTiles() %>% addMarkers(lat = 1.239660, lng = 103.835381,
             popup = "Sentosa Cove")
m
```





What about trying to plot a map showing Disneyland in Orlando US?

Singapore has many tourist attractions other than Sentosa Island, how do we add all of them to the map?

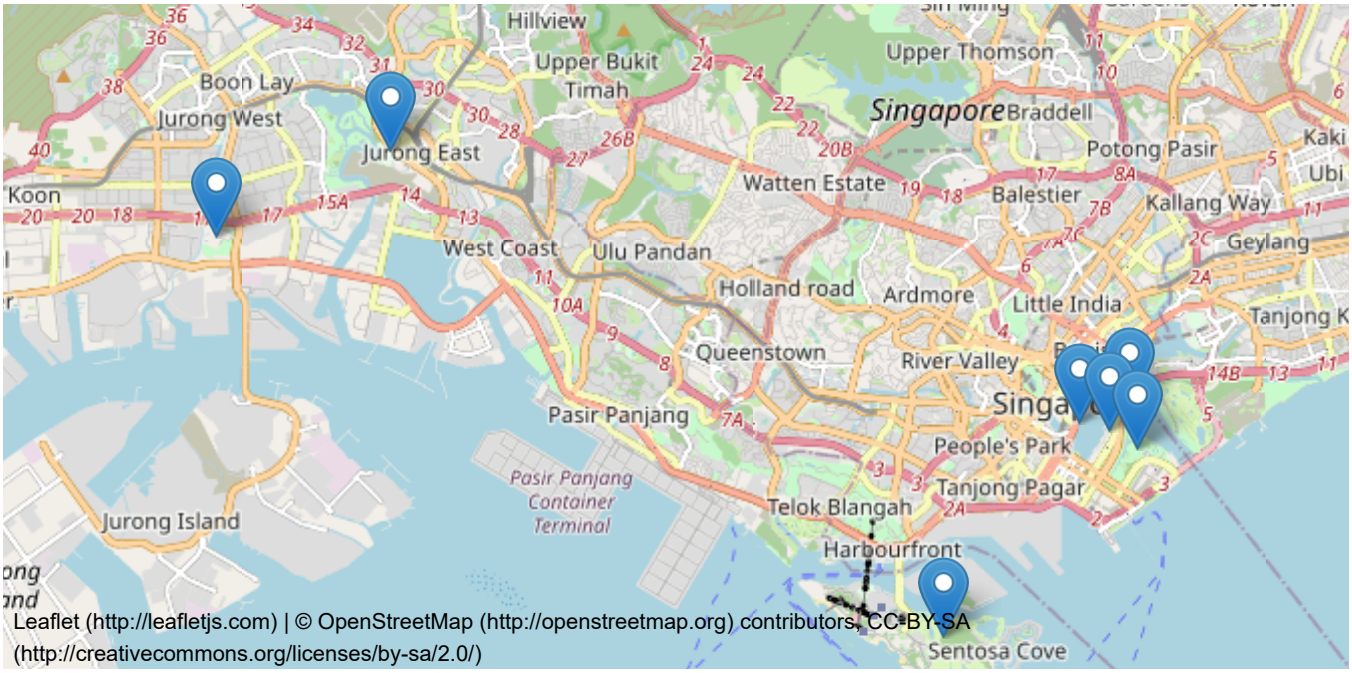
First, we need to create a dataframe that contains the longitude, latitude and the names of all tourist attractions.

```
df <- data.frame(latitude = c(1.25011, 1.28544, 1.318707,
                              1.282375, 1.4043, 1.4022,
                              1.4029, 1.2868, 1.3332, 1.2893),
                 longitude = c(103.83093, 103.859590, 103.706442,
                               103.864273, 103.7930, 103.7881,
                               103.7917, 103.8545, 103.7362,
                               103.8631),
                 name = c("Sentosa Island", "Marina Bay Sands",
                          "Jurong Bird Park",
                          "Gardens By the Bay", "Singapore Zoo",
                          "Night Safari", "River Safari",
                          "Merlion Park", "Science Center",
                          "Singapore Flyer"))
```

Now, we can use what we did for Sentosa Island previously but with a small change.

Supply df as the data source and the specify the mapping of longitude, latitude and popup message to the columns in df.

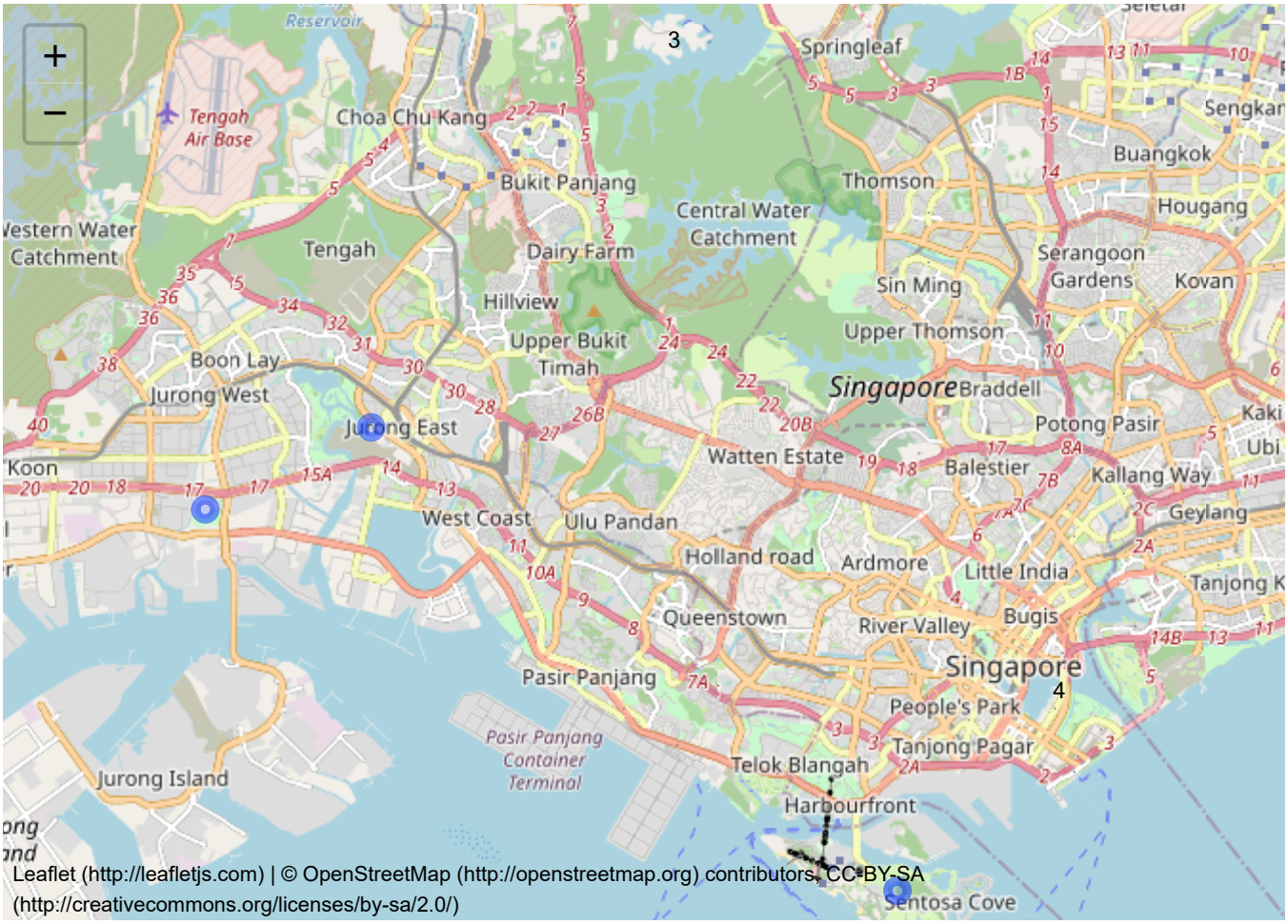
```
leaflet() %>%  addTiles() %>%
  addMarkers(data = df, lng = ~longitude, lat = ~latitude,
             popup = ~name)
```



Notice that many tourist attractions are concentrated around certain areas. It is very likely that a tourist will visit them all in one go. So, we should instead show them as clusters rather than individual spots.

We will use *addCircleMarkers()* for this goal.

```
leaflet() %>%  addTiles() %>%
  addCircleMarkers(data = df, lng = ~longitude, lat = ~latitude,
                   radius = 5,
                   clusterOptions = markerClusterOptions())
```



The places close by are now shown as just a circle with the number of places on it. Notice that the numbers will change as you zoom in and out.

Case Study: Pizza Hut

Similar to ggplot2, leaflet allows us to overlay geometric objects to form more complex views. More importantly, leaflet provides interactive controls that allow users to choose what layers to show on the map.

We will aim to have circle markers for each branch, and for them to have different colors depending on which region they belong to.

Let’s take a look at the pizza hut data again.

```
pizzahut.location
```

```
##                       Address Zipcode  Region         Location      lon
## 1                   Ang Mo Kio  560715   North Singapore 560715 103.8459
## 2                 Hougang Mall  538766   North Singapore 538766 103.8939
## 3                          Nex  556083   North Singapore 556083 103.8718
## 4                Thompson Plaza  574408   North Singapore 574408 103.8309
## 5                     Tao Payoh  310190   North Singapore 310190 103.8494
## 6  North Point Shopping Centre  769098   North Singapore 769098 103.8356
## 7                Causeway Point  738099   North Singapore 738099 103.7862
## 8              The Seletar Mall  797653   North Singapore 797653 103.7509
## 9                    Sun Plaza  757713   North Singapore 757713 103.8192
## 10              Waterway Point  828761   North Singapore 828761 103.7509
## 11                       Bedok  460215    East Singapore 460215 103.9335
## 12               Tampines Mall  529510    East Singapore 529510 103.9452
## 13                  Bedok Mall  467360    East Singapore 467360 103.7509
## 14                       OneKM  437157    East Singapore 437157 103.7509
## 15             East Point Mall  528833    East Singapore 528833 103.9530
## 16         Harbourfront Centre   99253 Central  Singapore 99253 103.7509
## 17                 Lucky Plaza  238863 Central Singapore 238863 103.8338
## 18               Marina Square   39594 Central  Singapore 39594 103.7509
## 19             Plaza Singapura  238839 Central Singapore 238839 103.8454
## 20            City Square Mall  208539 Central Singapore 208539 103.8566
## 21         Bukit Panjang Plaza  677743    West Singapore 677743 103.7642
## 22           Bukit Timah Plaza  588996    West Singapore 588996 103.7790
## 23               Chao Chu Kang  689812    West Singapore 689812 103.7450
## 24                   West Mall  658713    West Singapore 658713 103.7491
## 25                Jurong Point  648886    West Singapore 648886 103.7065
## 26                     Westgate  608532    West Singapore 608532 103.7509
##         lat
## 1  1.371011
## 2  1.372629
## 3  1.350644
## 4  1.354637
## 5  1.332654
## 6  1.429848
## 7  1.435855
## 8  1.345871
## 9  1.448417
## 10 1.345871
## 11 1.326057
## 12 1.352661
## 13 1.345871
## 14 1.345871
## 15 1.342806
## 16 1.345871
## 17 1.304360
## 18 1.345871
## 19 1.301016
## 20 1.311403
## 21 1.379962
## 22 1.339001
## 23 1.384848
## 24 1.349649
## 25 1.339874
## 26 1.345871
```

It contains the address, zip code, region, location, longitude, and latitude of all the Pizza Hut branches in Singapore.

Let's first create a vector of the four regions used in the data.

```
region.list <- c("North", "East", "Central", "West")
```

Next, we will create a colorfactor object. What this does is to map color codes to the regions in our data.

```
colorFactors <- colorFactor(c('red', 'green', 'blue', 'brown'),
                            domain = pizzahut.location$Region)
```

Then, we initiate the leaflet widget.

```
m <- leaflet() %>% addTiles()
```

Then, I will go through the four regions to add layers one by one. For each region, pizzahut.region contains data only belonging to the region. Then, we add a layer with circle markers defined in this pizzahut.region. Note that I added two important parameters: One is to specify the color code to be used for this region, and the other is to define a group parameter for this layer.

```
for(i in 1:4)
{
  pizzahut.region <- pizzahut.location[pizzahut.location$Region == region.list[i],]

  m <- addCircleMarkers(m,
                        lng=pizzahut.region$lon,
                        lat=pizzahut.region$lat,
                        popup=pizzahut.region$Address,
                        radius=10,
                        stroke = FALSE,
                        fillOpacity = 1,
                        color = colorFactors(pizzahut.region$Region),
                        group = region.list[i]
  )
}
```

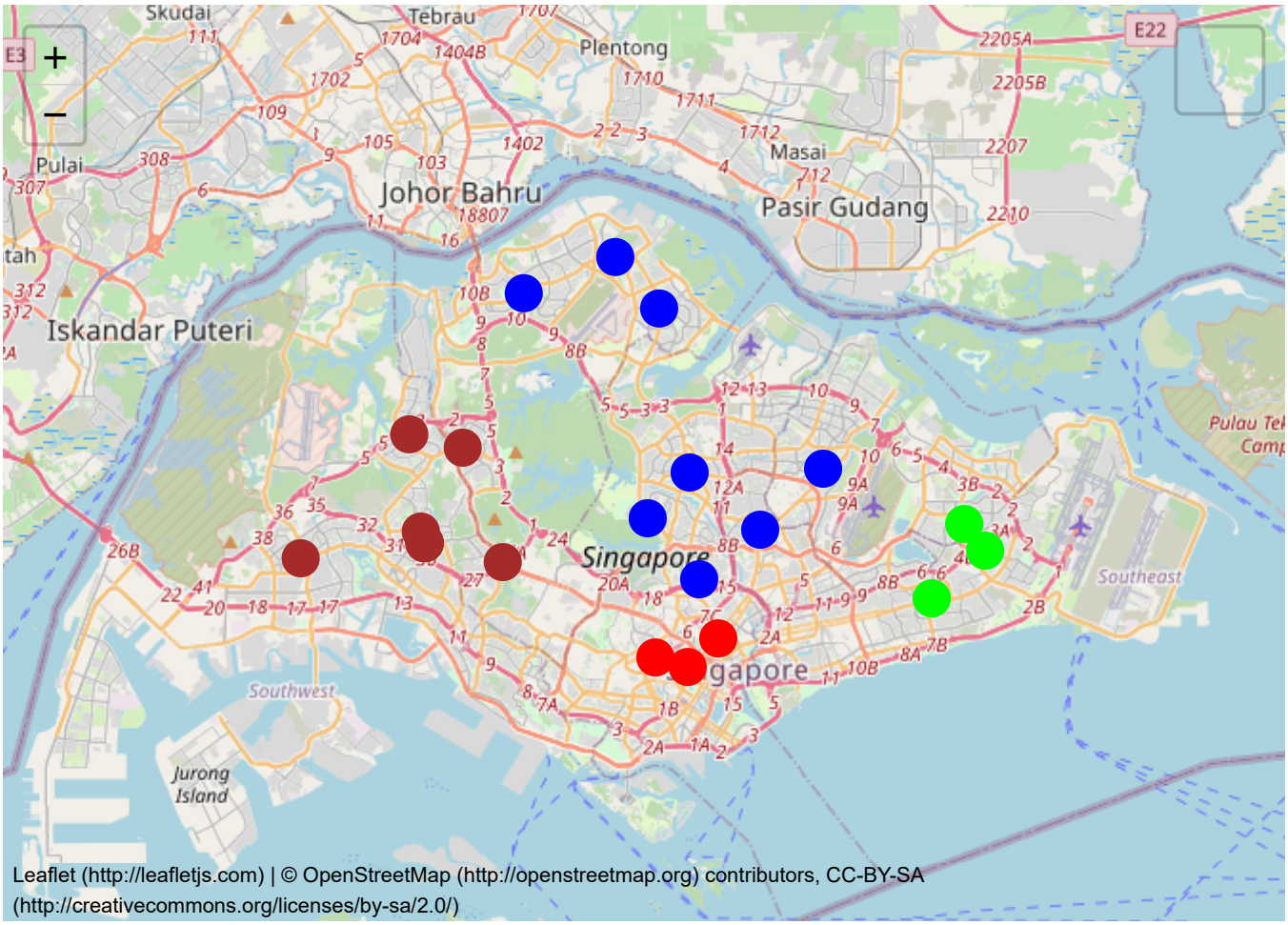
Next, I am going to add different types of maps. The names given in the functions are the provider names, but I selected four of them. You can visit this website (http://leaflet-extras.github.io/leaflet-providers/preview/) to see a full list of providers. Again, a group parameter is specified for each layer.

```
m <- addTiles(m,group="Default")
m <- addProviderTiles(m,"Esri.WorldImagery", group = "Esri")
m <- addProviderTiles(m,"Stamen.Toner", group = "Toner")
m <- addProviderTiles(m, "Stamen.TonerLite", group = "Toner Lite")
```

Finally, I will put all of them together as controls. We need to supply the names of groups defined in the previous functions here.

There are two types of groups, basegroups are the basic layer of the map. Every time, only one of these groups will be shown. That's why the controls eventually show as radio buttons. Overlaygroups can overlay with each other, meaning they can co-exist. Therefore, they will show as checkboxes.

```
m <- addLayersControl(m, baseGroups = c("Default","Esri",
                                        "Toner Lite","Toner"),
                      overlayGroups = region.list
)
m
```



Finally, you can save the result as an HTML page and embed it on your own website.

```
library(htmlwidgets)
saveWidget(m, file="pizzahutlocations.html")
```

Polygons

In leaflet, addMarkers or addCircleMarkers place markers based on locations of the data points. Sometimes, we may visualize data on a regional basis.

For example, we may want to compare annual rainfall over five region(s) of Singapore. The denser a region is colored, the greater the rainfall it has. The coloring is not perfect here due to the poor quality of the region boundary data.

The following codes show how we can obtain the map data and integrate it with simple made-up data representing rainfall over regions. Go to the Map View by Borders section for a recap on what these commands do.

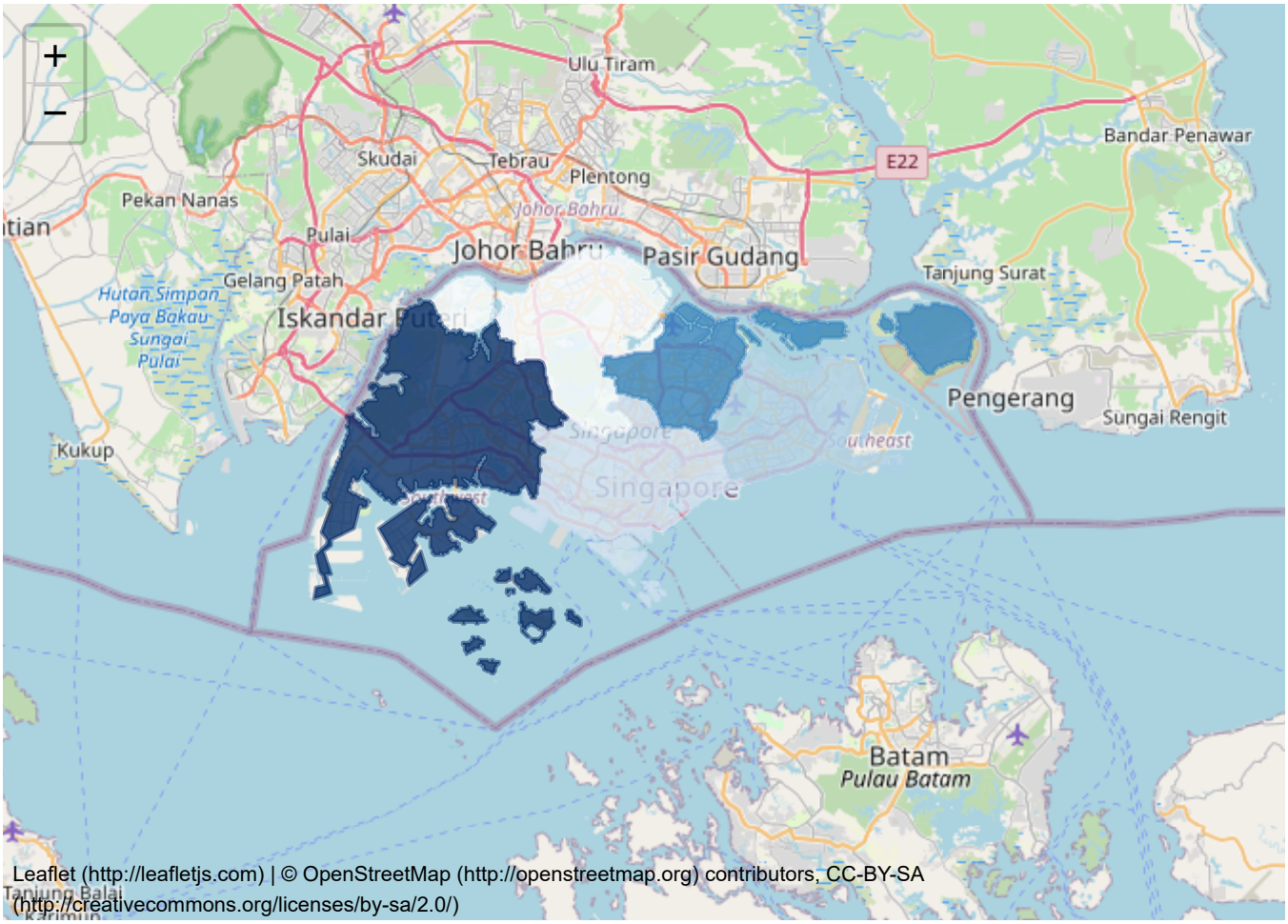
```
library(raster)

data_1 <- data.frame(Region = c('Central','East','North','North-East','West'),
                     Value = c(3,4,1,7,10))

SG<-getData('GADM', country='SG', level=1)

SG@data <- merge(SG@data,data_1,by.x="NAME_1",by.y="Region")

popup <- paste0("<strong>Name: </strong>",SG$NAME_1)
```

Next, we create a palette with blue as the key color scheme and the color sensitivity proportionated by the value of data.

```
pal <- colorNumeric( palette = "Blues", domain = SG$Value)
```

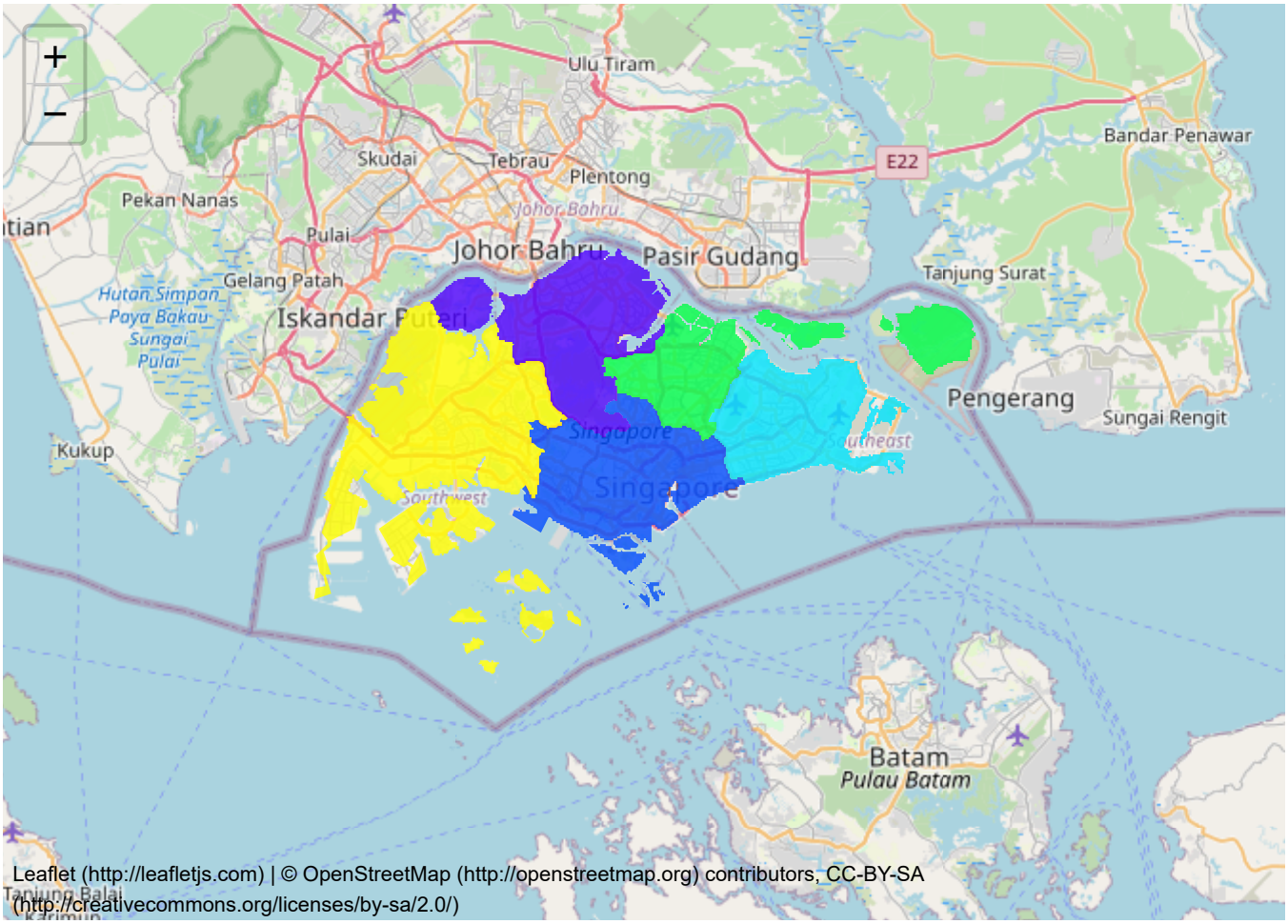
We are now ready to produce the map.

```
#continuous palette color
m<-leaflet() %>% addTiles() %>% addPolygons(data=SG, weight = 2,
        stroke = TRUE, smoothFactor = 0.1, fillOpacity = 0.8, color = ~pal(Value), popup=popu
p)

m
```



Sometimes, we may just want to differentiate the regions by different colors. It can be done by using a discrete color setting.

```
#random color
factpal <- colorFactor(topo.colors(5), SG$Value)

m<-leaflet() %>% addTiles() %>% addPolygons(data=SG, weight = 2,
  stroke = FALSE, smoothFactor = 0.2, fillOpacity = 0.8, color = ~factpal(Value), popup=popu
p)

m
```



Again, you can save the result as an HTML page and embed it on your own website.

```
library(htmlwidgets)
saveWidget(m, file="attractions.html")
```

Heatmaps

We can compare data over regions by polygon coloring, in which each region is colored uniformly. We can create a heatmap where the density of the color is based on the density of events at locations.

Here is a simple method to create such a Heatmap. This is developed based on the codes shared on this website (https://rpubs.com/bhaskarvk/leaflet-heatmap).

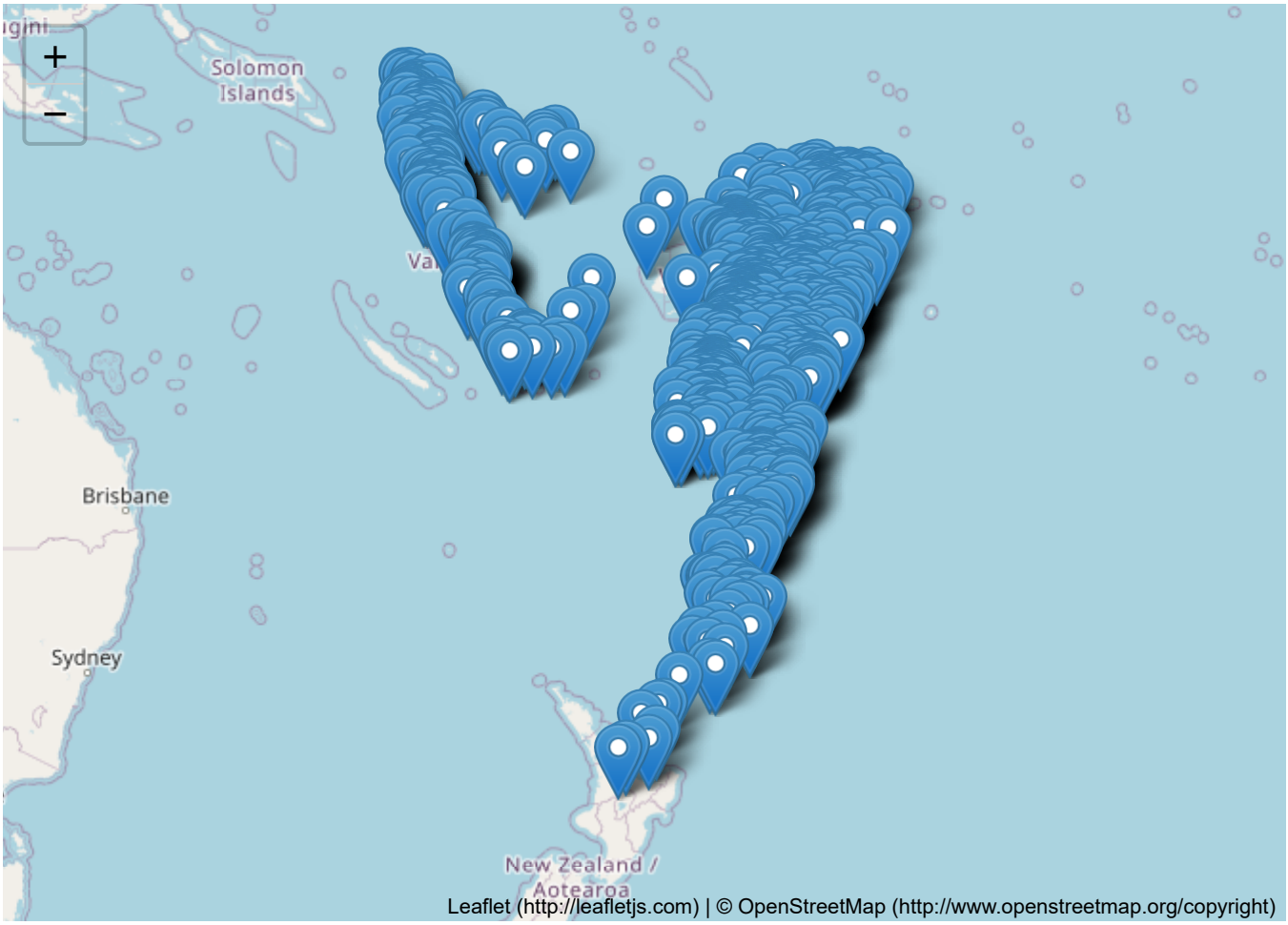
R has a built-in dataset that contains a series of earthquakes that occurred near Fiji.

Firstly, you need to install a library called “leaflet.extras”.

```
library(leaflet.extras)

#Some version of RStudio have problems rendering images if the source of the map is on https
#So set this option to direct the view on the web browser
options("viewer" = function(url, ...) utils::browseURL(url))
```

Let's first take a look at how the earthquakes are distributed, using the *addMarker()* function.

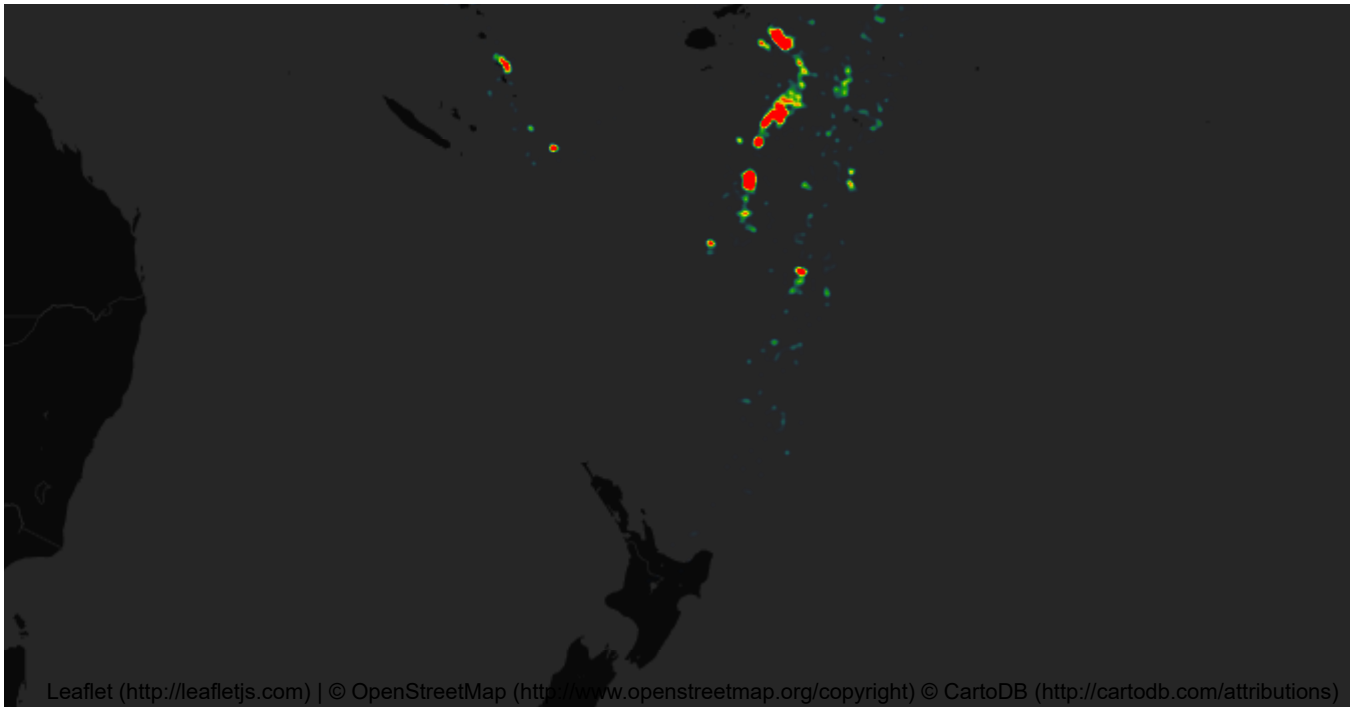
```
#Use markers to mark all the earthquakes on the map
leaflet(quakes) %>% addProviderTiles(providers$OpenStreetMap.Mapnik) %>%
  addMarkers(lng = ~long , lat = ~lat)
```



This is a bit messy as the markers are overlapping with each other.

To generate a heatmap, just use the following command:

```
#Now create heat map that "heat" is based on the number of earthquakes
#around different locations.
m<-leaflet(quakes) %>% addProviderTiles(providers$CartoDB.DarkMatter) %>%
  addWebGLHeatmap(lng=~long, lat=~lat, size = 60000)
m
```



To provide better contrast, we used a dark map background. You can also change the provider to see different effects. The default size is measured in meters. Above we have each event's physical radius being 60,000 meters.

We can also color the heatmap based on the earthquake's magnitudes instead of in the number of occurrences. This can be done by supplying the magnitude as the intensity.

Again, you can save the result as an HTML page and embed it on your own website.

```
library(htmlwidgets)
saveWidget(m, file="heatmap.html")
```

Spatial Data Visualisation

- 1. Creating spatial data visualisation using the **raster** Package
- 2. Using the **rnaturalearth** Package
- 3. Using the **urbanmapr** Package
- 4. Using the **tigris** Package
- 5. Plotting maps using the **tmap** Package
- 6. Loading shapefiles downloaded online for spatial visualisation

1. Creating spatial data visualisation using the **raster** Package

One package used to create spatial data visualisations is the **raster** package, with the **getData()** function used to obtain spatial data, before plotting it with **ggplot2** or other packages such as leaflet. In this example, we will be looking to plot a map of france, highlighting deaths due to COVID-19 across the regions.

```
library(tidyverse)
library(raster) # For the "getData" function, Full list of countries are available at http
s://gadm.org/maps.html
library(ggplot2)
library(tigris) # For the "geo_join" function
library(scales) # For the "breaks_pretty" function
library(rgdal) # For the "spTransform" function
library(broom) # For the "tidy" function.
```

```
france <- getData('GADM', country = 'FRA', level = 1)

france <- spTransform(france, CRS("+init=EPSG:32631")) # To make map projections more accurat
e
```

Using the **getData()** function, countries are identified by their ISO code, with a list here (https://www.nationsonline.org/oneworld/country_code_list.htm).
level = 0 gives you country outline, and level = 1 gives you regional boundaries.
If you are looking for the most accurate visualisation, you should use spTransform to reproject the area you are going to plot to make the resulting visualisation more accurate. To perform this step:
1) Obtain the longitude and latitude of the location you are looking to plot (either online or with a package).
2) Enter the longitude and latitude here (http://www.dmap.co.uk/ll2tm.htm), and convert the coordinates to a grid reference using the “UTM (WGS84)” Grid Area. With the coordinates 46.2276° N, 2.2137° E, the resulting grid number would be 31. Your grid square would be N, based on your latitude, giving you a resulting UTM grid of 31N.
3) In this Spatial Reference Website (https://spatialreference.org/ref/epsg/), enter your UTM grid in the search references, and you will see a list of EPSG references. Copy the one used for WGS84 grid, and that goes into the CRS argument of the spTransform function, as seen above.

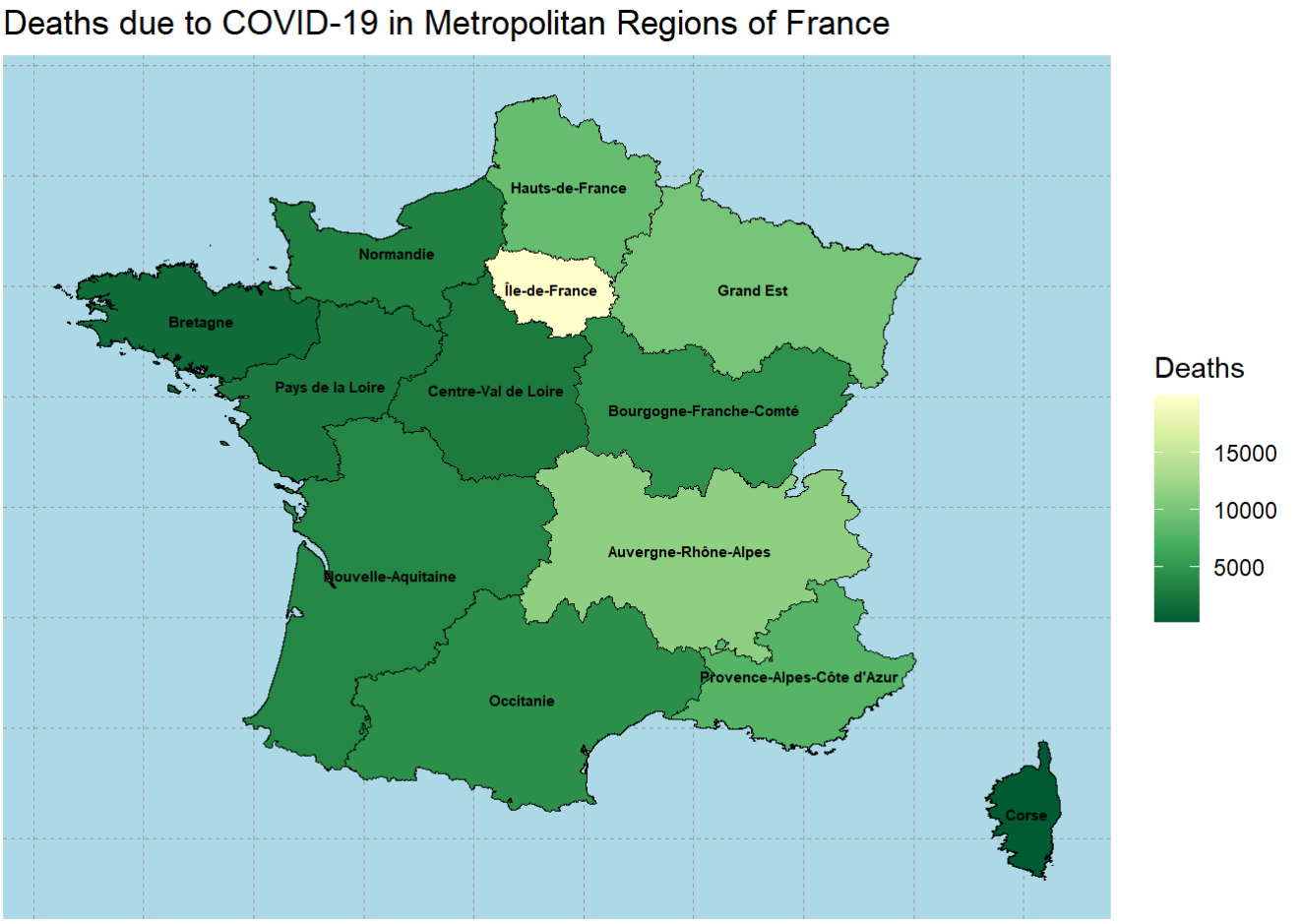
```
# Data cleaning to obtain our desired covid dataset

# Function to keep regions that are only available in the getData function
seek <- function(x, y) {
  return(any(grepl(x,y,ignore.case = TRUE)))
}

# To obtain data from Metropolitan French regions as defined in the getData function, with ma
tching names
covid <- read.csv("jrc-covid-19-all-days-by-regions.csv") # Covid -19 dataset
covidfr <- covid[covid$CountryName == "France",] # Including only data from france
covidfr$Region <- gsub("Auvergne-RhÃ´ne-Alpes", "Auvergne-Rhône-Alpes", covidfr$Region)
covidfr$Region <- gsub("ÃŽle-de-France", "Île-de-France", covidfr$Region)
covidfr$Region <- gsub("Bourgogne-Franche-ComtÃ©", "Bourgogne-Franche-Comté", covidfr$Region)
covidfr$Region <- gsub("Provence-Alpes-CÃ´te d'Azur", "Provence-Alpes-Côte d'Azur", covidfr$R
egion)
covidfr <- covidfr[covidfr$Region != "",] # Removing blank regions
covidfr <- covidfr[sapply(covidfr$Region, seek, france@data$NAME_1),]
# To obtain cumulative number of deaths per region in France
covidfr <- covidfr %>% group_by(Region) %>% summarise(deaths = max(CumulativeDeceased))

## To combine covid data with spatial data previously obtained, and converting the result to
 a dataframe
france@data$id <- rownames(france@data)
france_merged <- geo_join(france, covidfr, "NAME_1", "Region") # The merge function can be us
ed as well
# tidy is used to convert spatial dataframe to a dataframe to be used for geom_polygon
france_df <- tidy(france_merged)
```

```
## Regions defined for each Polygons
```

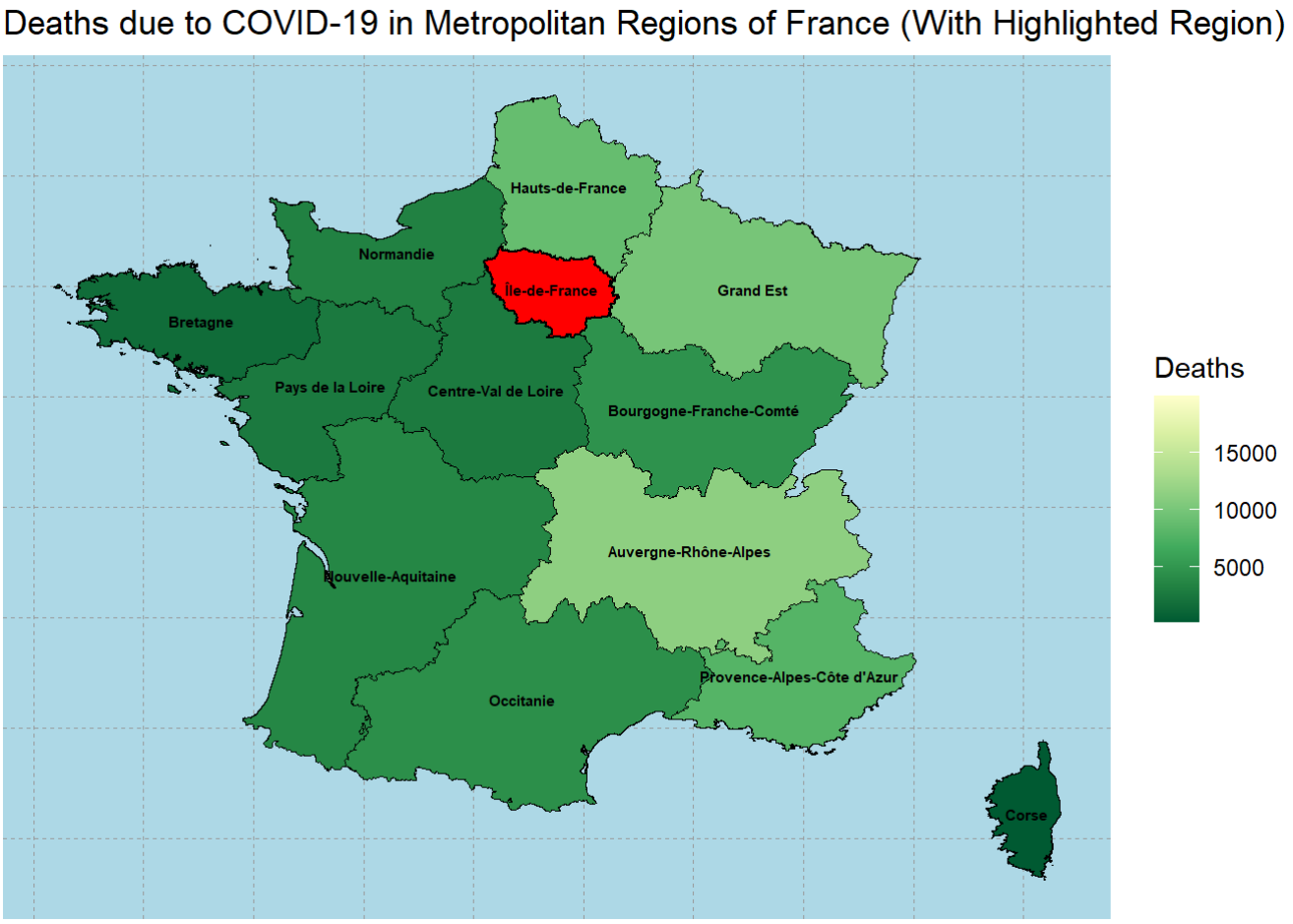
```
france_df <- left_join(france_merged@data,france_df, by = "id")


centroid_df <- as.data.frame(coordinates(france))
centroid_df <- cbind(centroid_df, unique(france_df$NAME_1))
colnames(centroid_df) <- c("long","lat","Region")

# To standardise plot themes
theme_opts<-list(theme(panel.grid.minor = element_line(color = "gray60", linetype = "dashed",
size = 0.25),
                       panel.grid.major = element_line(color = "gray60", linetype = "dashed",
size = 0.25),
                       panel.background = element_rect(fill = "lightblue"),
                       plot.background = element_blank(),
                       axis.line = element_blank(),
                       axis.text.x = element_blank(),
                       axis.text.y = element_blank(),
                       axis.ticks = element_blank(),
                       axis.title.x = element_blank(),
                       axis.title.y = element_blank()))

# To highlight one specific region (The region with the highest number of deaths in this cas
e)
covidfr_ordered <- order(covidfr$deaths, decreasing = TRUE)
highest_deaths_fr <- head(covidfr[covidfr_ordered,], 1)
highest_region <- france[france@data$NAME_1 == highest_deaths_fr$Region,]
highest_region <- tidy(highest_region)
```

```
## Regions defined for each Polygons
```

```
# Plotting the various Spatial Data Visualisations

ggplot() +
  geom_polygon(data = france_df, aes(x = long,y = lat, group = group, fill = deaths), color =
"black", size = 0.25) +
  geom_text(data = centroid_df, aes(label = Region, x = long, y = lat, fontface = "bold"), co
l = "black", size = 2) +
  theme_opts +
  scale_fill_distiller(name="Deaths", palette = "YlGn", breaks = breaks_pretty(n = 3)) +
  ggtitle("Deaths due to COVID-19 in Metropolitan Regions of France")
```




```
# Region with greatest number of deaths is highlighted here
ggplot() +
  geom_polygon(data = france_df, aes(x = long,y = lat, group = group, fill = deaths), color =
"black", size = 0.25) +
  # To highlight region with largest number of deaths
  geom_polygon(data = highest_region, aes(long,lat,group=group), fill = "red", color = "blac
k") +
  geom_text(data = centroid_df, aes(label = Region, x = long, y = lat, fontface = "bold"), co
l = "black", size = 2) +
  theme_opts +
  scale_fill_distiller(name="Deaths", palette = "YlGn", breaks = breaks_pretty(n = 3)) +
  ggtitle("Deaths due to COVID-19 in Metropolitan Regions of France (With Highlighted Regio
n)")
```



2. Using the **rnaturalearth** Package

Another package commonly used to create spatial data visualisations is the **rnaturalearth** package to obtain shape files. One advantage of the rnaturalearth package over the raster package is that it allows users to obtain spatial data for the world map and the different continents as well, instead of just countries.

```
library(rnaturalearth)
library(rnaturalearthhires) # Required for scale = "large"
```

In the rnaturalearth package, pre-downloaded maps can be accessed with:
1. **ne_countries()** for country (admin-0) boundaries
2. **ne_states()** for boundaries within countries (admin-1)
3. **ne_coastline()** for world coastline

```
# Includes countries, dependencies, territories and islands as well
world <- ne_countries(scale = "large", returnclass = "sf")
```

To see all countries, dependencies, territories and islands covered by this package (Subjected to change as the package is updated):

```
library(DT) # Used to obtain the datatable below
All_locations <- data.frame(Locations = world$admin)
datatable(All_locations, rownames = FALSE, options = list(pageLength = 10, lengthChange = F))
```

Search:

Locations
Indonesia
Malaysia
Chile
Bolivia
Peru
Argentina
Dhekelia Sovereign Base Area
Cyprus
India
China

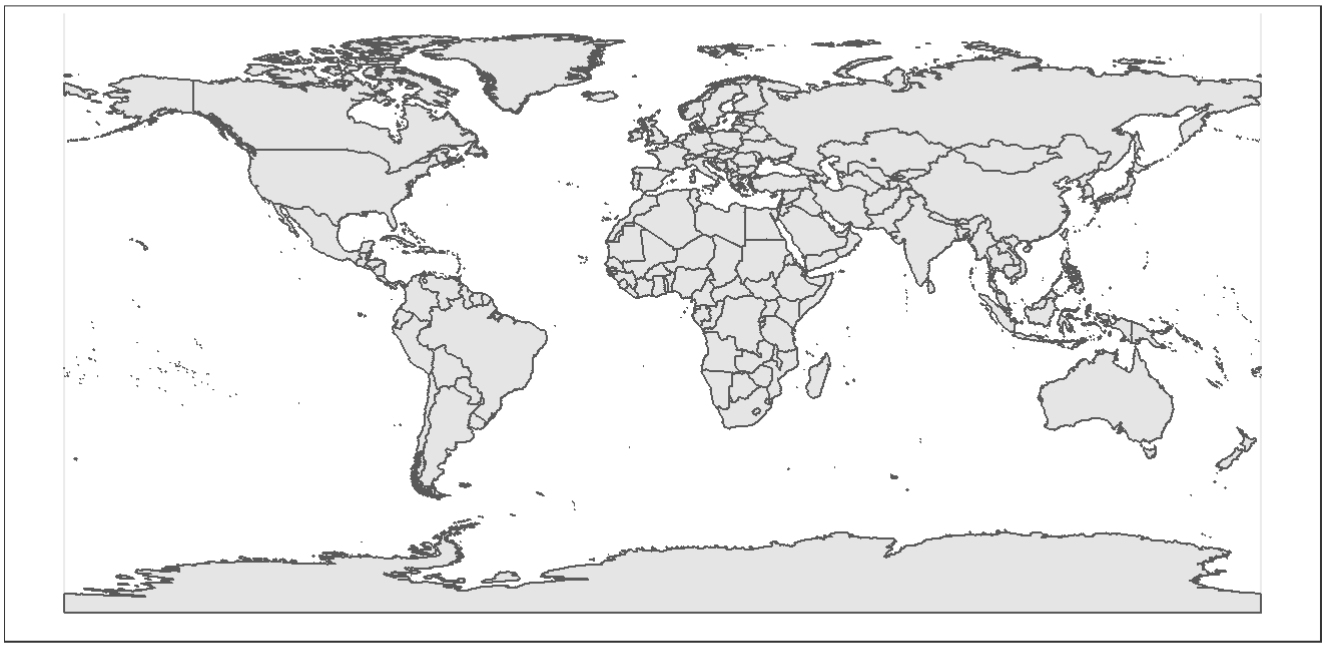
Showing 1 to 10 of 255 entries Previous 1 2 3 4 5 … 26 Next

We're using **returnclass = "sf"** here to plot spatial data visualisations using **"sf"** class instead of **returnclass = "sp"**, which returns a "SpatialPolygonsDataFrame" or **"sp"** class, just like the **getData()** function used earlier. That way, we can plot our data with **geom_sf()** instead of **geom_polygon()**, which might be more convenient since it already takes into consideration the coordinate system, and allows us to easily choose the area we wish to show on our plot using the **coord_sf()** function. Also, we no longer have to use the **tidy()** function to convert our data into a dataframe.

Large scaled data is the most detailed, followed by medium and small for the "scale" argument.

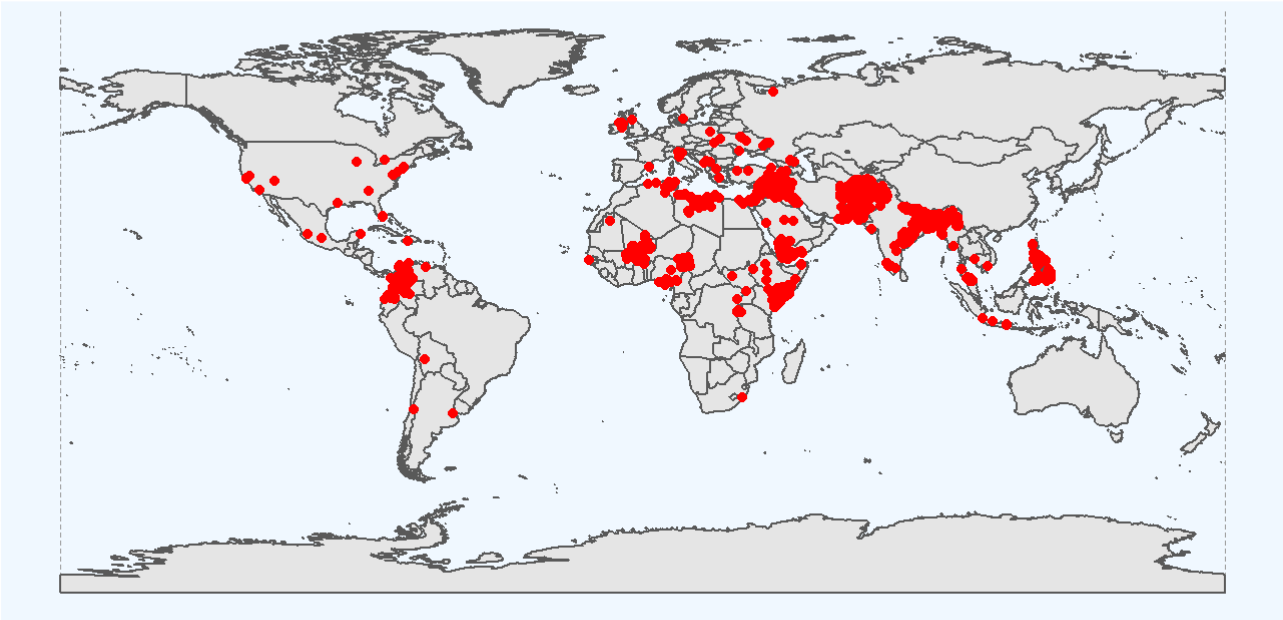
```
# Plotting a map of the world displaying bombing cases in 2018
ggplot() +
  geom_sf(data = world) +
  ggtitle("World Map") +
  theme_bw()
```

World Map



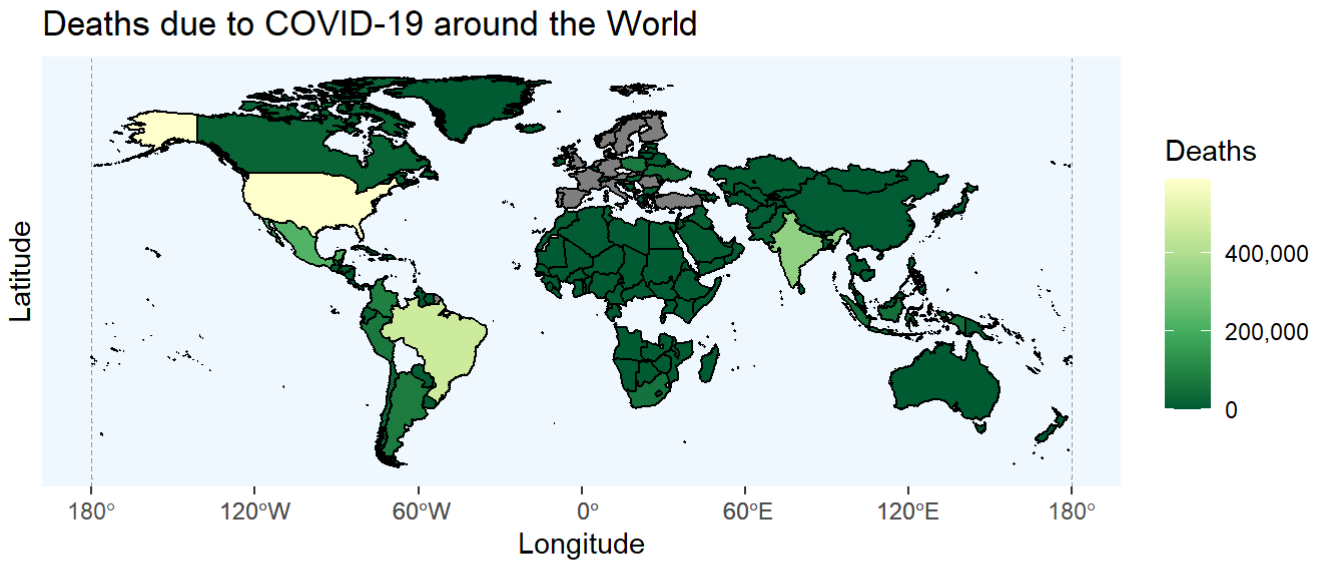
```
terrorism <- read.csv("globalterrorismdb_0919dist.csv")
terrorism <- terrorism[terrorism$attacktype1_txt == "Bombing/Explosion" & terrorism$iyear ==
2018,]

ggplot() +
  geom_sf(data = world) +
  geom_point(data = terrorism, aes(x = longitude, y = latitude), color = "red") +
  ggtitle(label = "Bombing incidents Around the World in 2018", subtitle = "Obtained from the
Global Terrorism Database") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  # This line of code below can be altered and included to manipulate the plotted area accord
ing to your desired       coordinates
  # coord_sf(xlim = c(100.00, 160.00), ylim = c(-45.00, -10.00), expand = FALSE) +
  xlab("") +
  ylab("")
```

```
## Warning: Removed 11 rows containing missing values (geom_point).
```

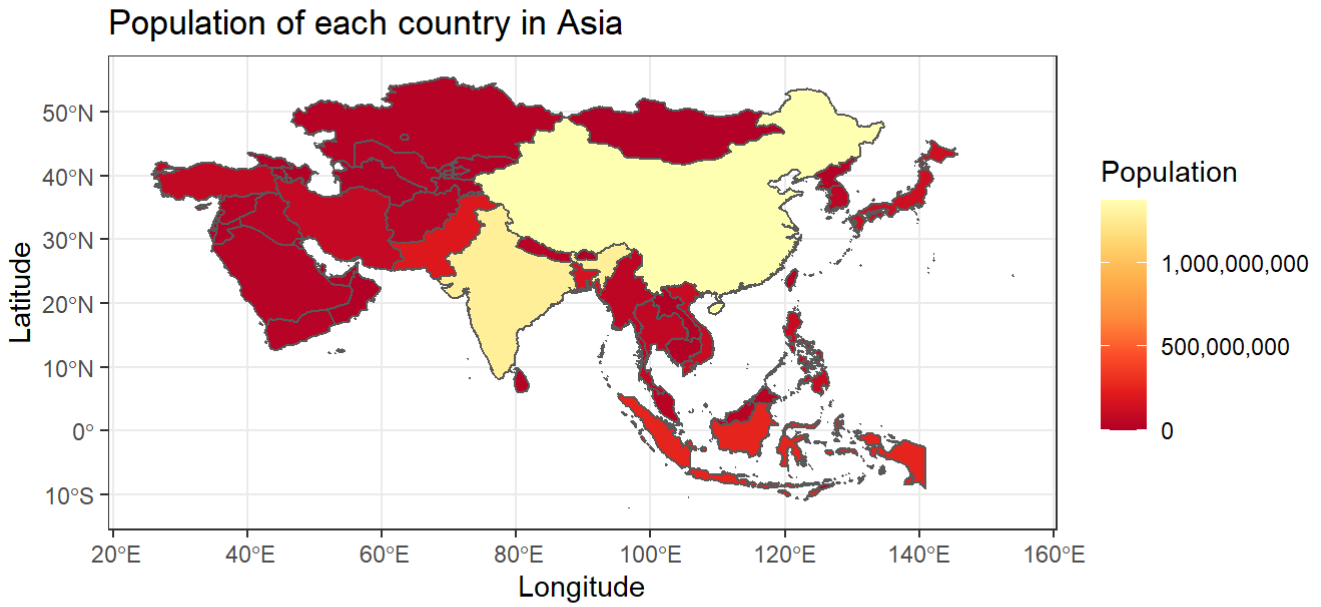


Similar to the first map showing deaths due to COVID-19 in France, we can incorporate covid data from the same dataset into the spatial data obtained using **ne_countries()**, and plot a map showing deaths due to COVID-19 around the world/continent

```
# Plotting deaths due to COVID-19 around the world, using ne_countries()

covidworld <- read.csv("jrc-covid-19-all-days-by-regions.csv")
covidworld <- covid %>% group_by(CountryName) %>% summarise(deaths = max(CumulativeDeceased))
covidworld_df <- merge(world, covidworld, by.x = "admin", by.y = "CountryName")

ggplot() +
  geom_sf(data = covidworld_df, aes(fill = deaths), color = "black") +
  scale_fill_distiller(name="Deaths", palette = "YlGn", breaks = breaks_pretty(n = 3), labels
= comma) +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Deaths due to COVID-19 around the World")
```

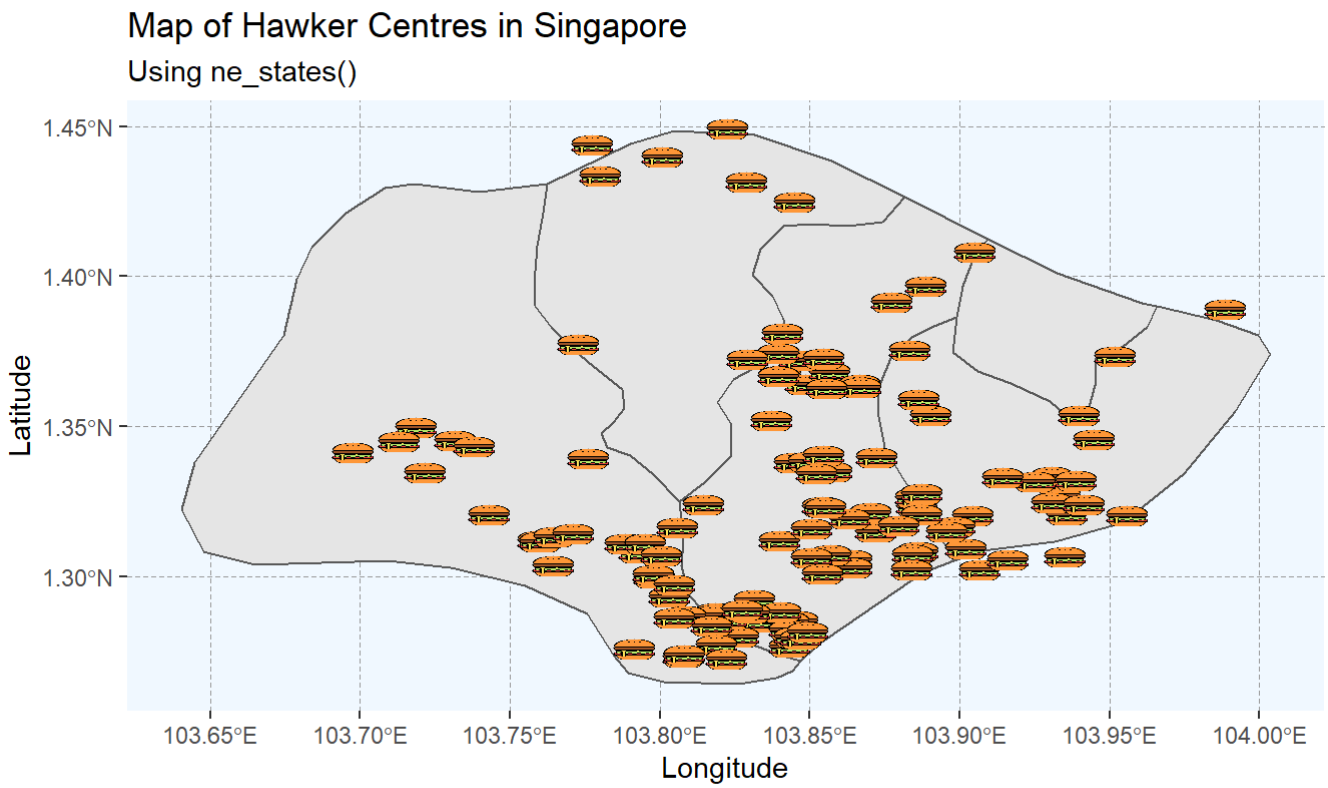



```
# Plotting a map of Asia displaying the population sizes of each country
asia <- ne_countries(scale = "large", continent = "asia", returnclass = "sf")

ggplot() +
  geom_sf(data = asia, aes(fill = as.numeric(pop_est))) + # Population data is provided in th
e rnaturalearth package
  scale_fill_distiller(name="Population", palette = "YlOrRd", breaks = breaks_pretty(n = 3),
 labels = comma) +
  labs(x = "Longitude", y = "Latitude") +
  ggtitle("Population of each country in Asia") +
  # coord_sf(xlim = c(100.00, 160.00), ylim = c(-45.00, -10.00), expand = FALSE) +
  # coord_sf(xlim = c(-180.00, 0.00), expand = FALSE) +
  theme_bw() +
  theme(aspect.ratio = 0.5)
```



rnaturalearth can be used to plot individual country maps as well, but as you can see from the example below, it might be more visually appealling to plot country maps using the **getData()** function from the raster package instead.

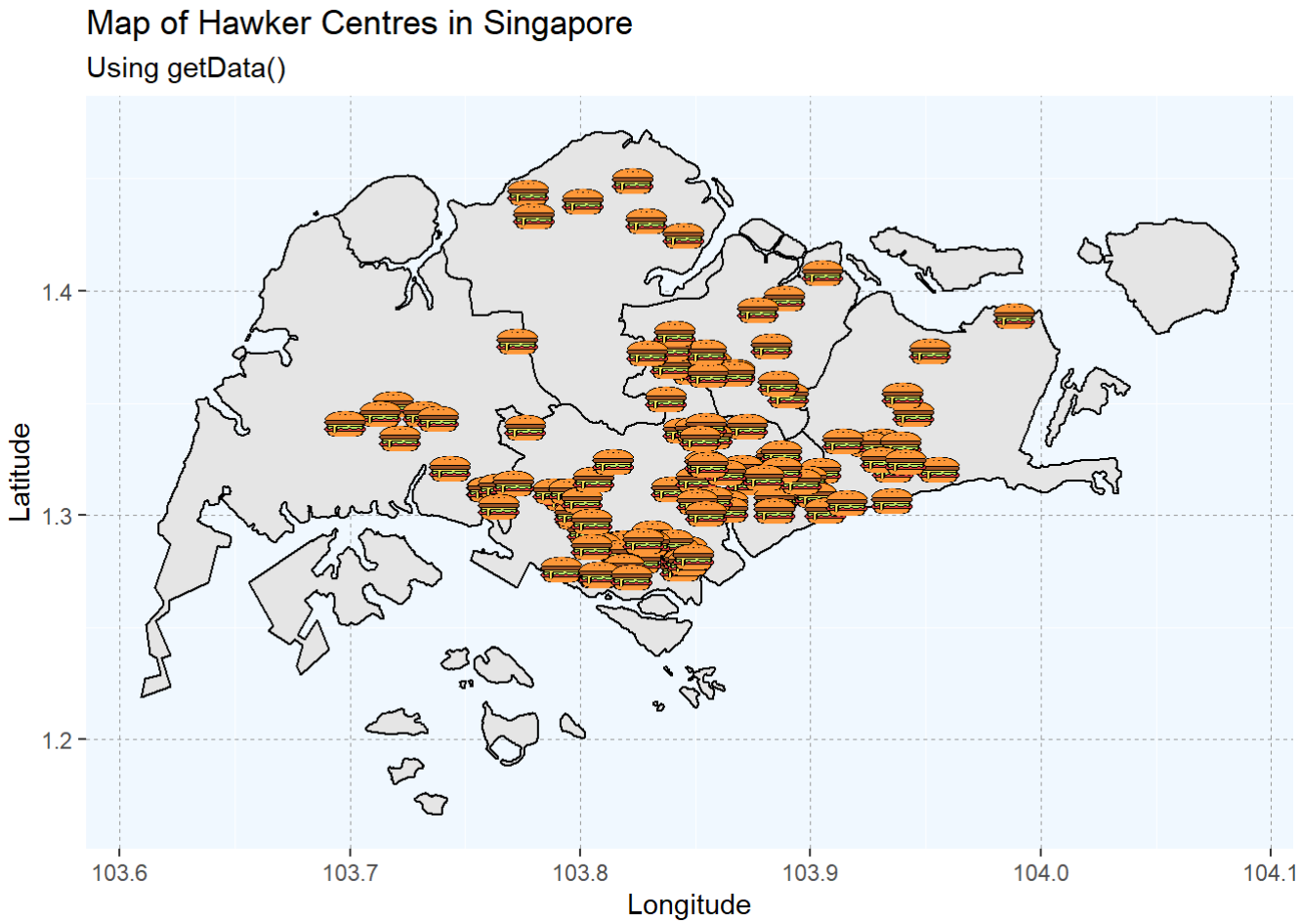
```
library(ggimage)
# Plotting a map of Singapore displaying all hawker centres using ne_states()

singapore <- ne_states(country = "singapore", returnclass = "sf")
hawkers <- read.csv("hawker-centres-kml.csv")
image <- "https://image.flaticon.com/icons/png/512/3075/3075977.png"
hawkers <- cbind(hawkers,image)

ggplot() +
  geom_sf(data = singapore) +
  geom_image(data = hawkers, aes(x = X, y = Y, image = image), size = .035) +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Map of Hawker Centres in Singapore", subtitle = "Using ne_states()")
```




```
SG <- getData('GADM', country = 'SG', level = 1)
SG <- tidy(SG)
```

```
## Regions defined for each Polygons
```

```
ggplot() + geom_polygon(data = SG, aes(x = long, y = lat, group = group), fill = "grey90", co
lor = "black") +
  xlab("Longitude") +
  ylab("Latitude") +
  #To show that images can be used to replace geom_point
  # geom_point(data = hawkers, aes(x = X, y = Y), color = "red") +
  geom_image(data = hawkers, aes(x = X, y = Y, image = image), size = .035) +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  ggtitle("Map of Hawker Centres in Singapore", subtitle = "Using getData()")
```

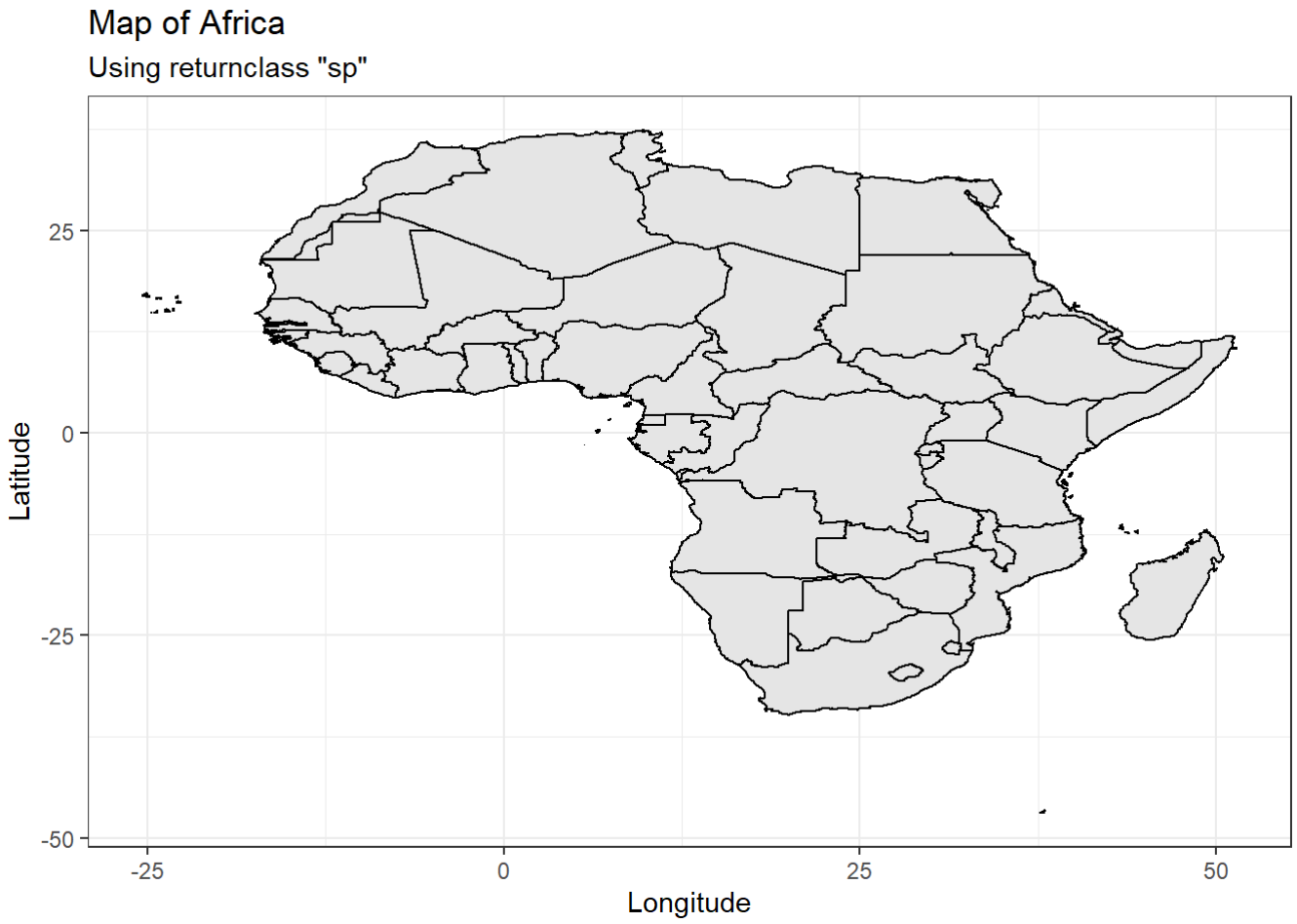


To show how we can plot the same map using both arguments of returnclass:

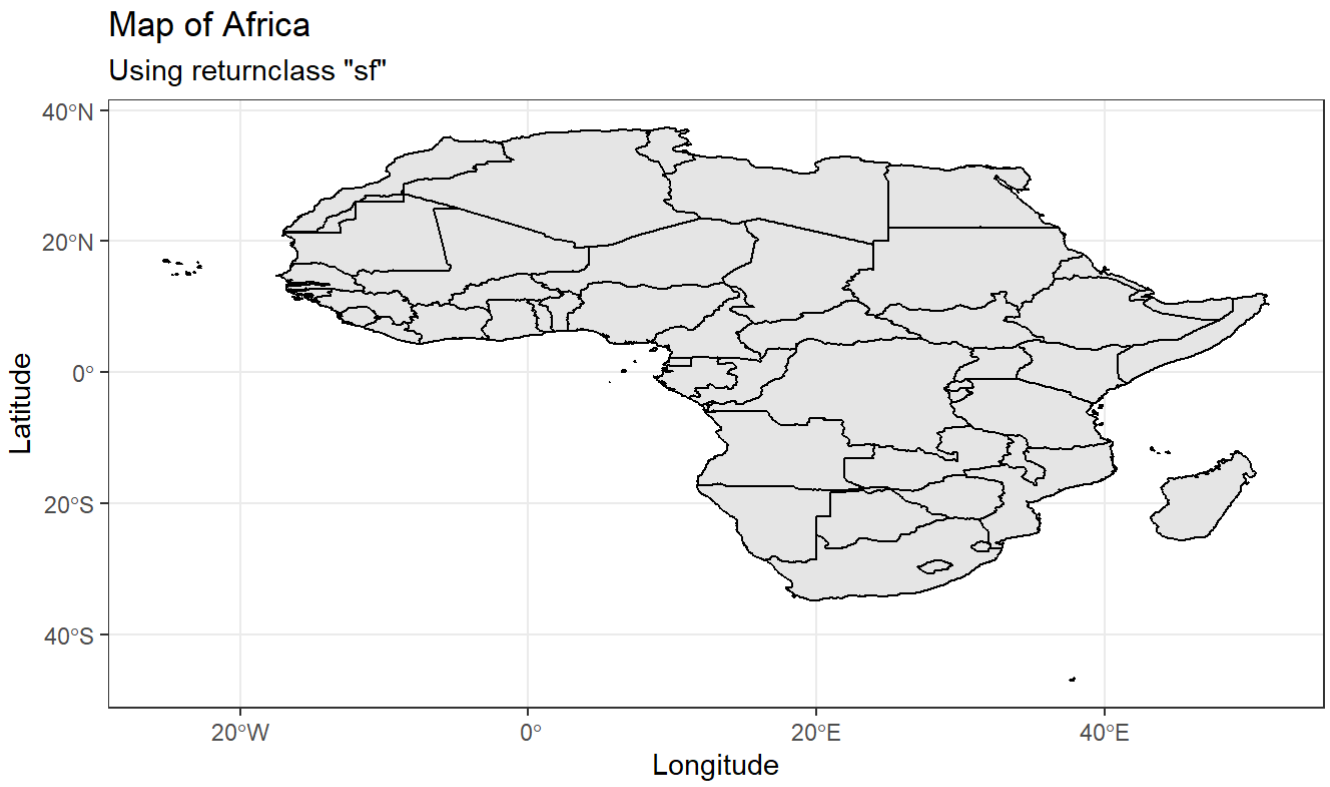
```
# To plot using returnclass = "sp" and geom_polygon()

# Does not need to be specified if using returnclass = "sp"
africa_sp <- ne_countries(scale = "large", continent = "africa", returnclass = "sp")
africa_sp <- tidy(africa_sp)
```

```
## Regions defined for each Polygons
```

```
ggplot() +
  geom_polygon(data = africa_sp, aes(x = long, y = lat, group = group), fill = "grey90", colo
r = "black") +
  labs(x = "Longitude", y = "Latitude") +
  theme_bw() +
  ggtitle("Map of Africa", subtitle = "Using returnclass \"sp\"")
```




```
# To plot the same map using returnclass = "sf"
africa_sf <- ne_countries(scale = "large", continent = "africa", returnclass = "sf")

ggplot() +
  geom_sf(data = africa_sf, fill = "grey90", color = "black") +
  xlab("Longitude") +
  ylab("Latitude") +
  theme_bw() +
  theme(aspect.ratio = 0.5) + # To make the plot look similar to the above, or it would look
 thinner
  ggtitle("Map of Africa", subtitle = "Using returnclass \"sf\"")
```



3. Using the **urbanmapr** Package

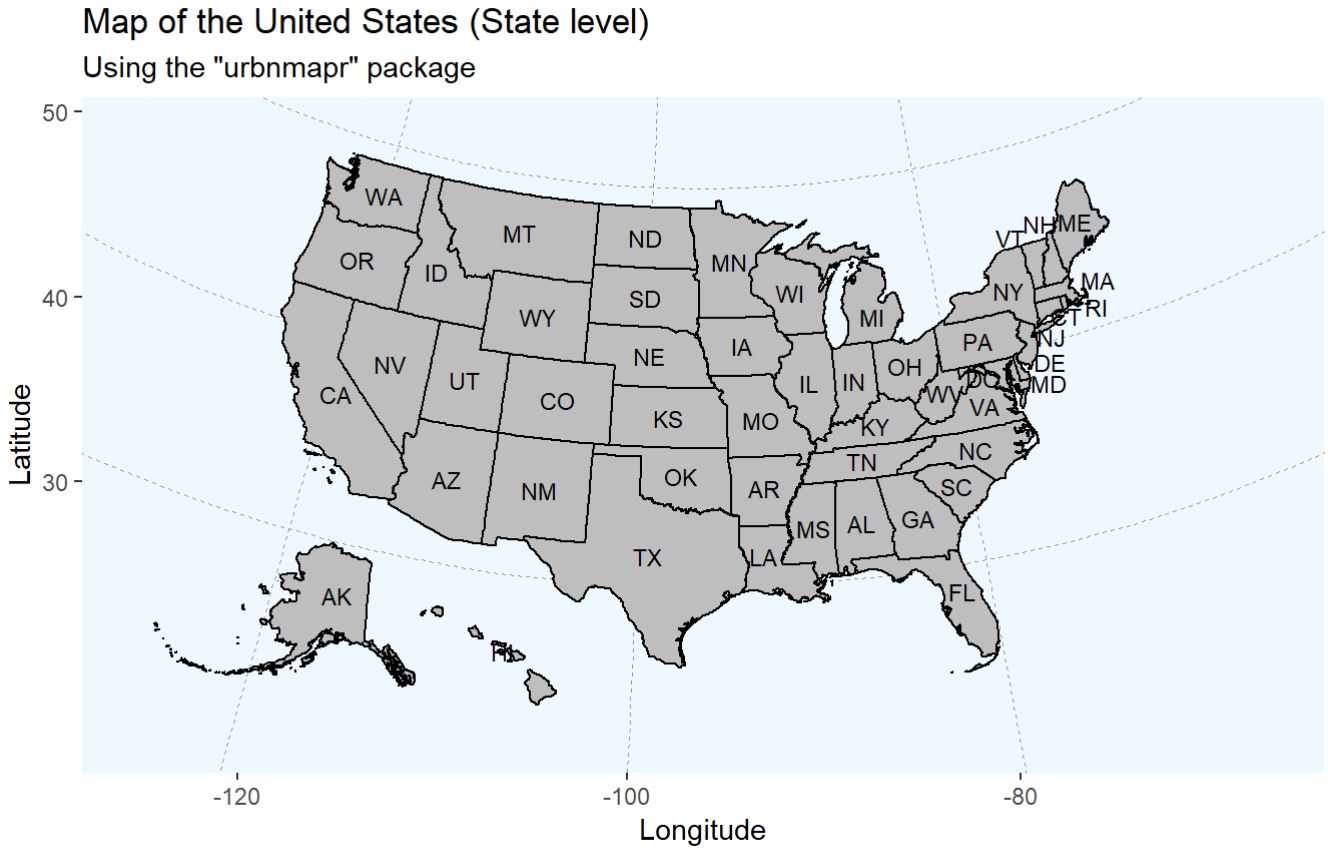
There is also a package **urbnmapr** that can be used specifically for obtaining spatial data about the United States. The benefit of this package is that Hawaii and Alaska are transformed to be displayed as insets within the continental United States, which might not be the case for other packages. The data provided in this package also requires minimal work to be merged with external datasets when creating your own visualisations.

However, before you can run the code below to install the package, you'll have to install **rtools 4.0** and follow the instructions provided here. (https://cran.r-project.org/bin/windows/Rtools/) Afterwards, you may install the package by running the code below, after installing the **devtools** package first.

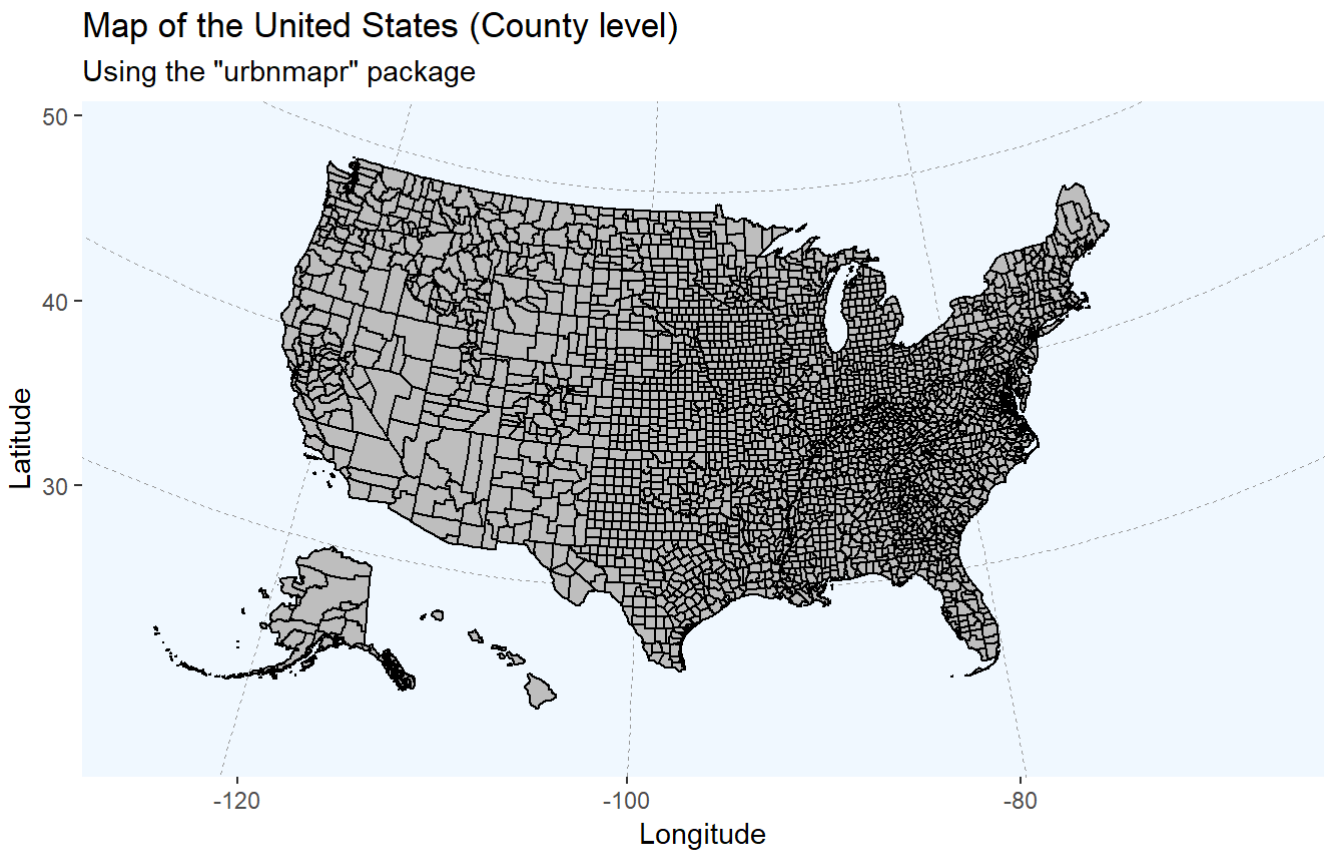
```
library(devtools)
install_github("UrbanInstitute/urbnmapr")
library(urbnmapr)
```

From there, we can either plot a map of the United States at states level (**urbnmapr::states**), or county level (**urbnmapr::counties**) using spatial data provided in the package. The **get_urbn_labels()** function also allows you to obtain centroid coordinates, which can be used to add labels to the map. In this document, it is necessary to specify the package that we use the "states" and "counties" functions from, as the tigris package already loaded to use the geo_join() function also contains functions that have the same names, and can be used to plot spatial data about the Unitd States as well.

```
# plotting at state level
states_label_data <- get_urbn_labels() # Default argument for map is "states", so it is unnce
ssary to                                                    specify this

ggplot() +
  geom_polygon(data = urbnmapr::states, mapping = aes(x = long, y = lat, group = group),
                   fill = "grey", color = "black") +
  geom_text(data = states_label_data, aes(label = state_abbv, x = long, y = lat), size = 3) +
  coord_map(projection = "albers", lat0 = 39, lat1 = 45) + # Used to alter map projection for
a nicer view
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Map of the United States (State level)", subtitle = "Using the \"urbnmapr\" packag
e") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```

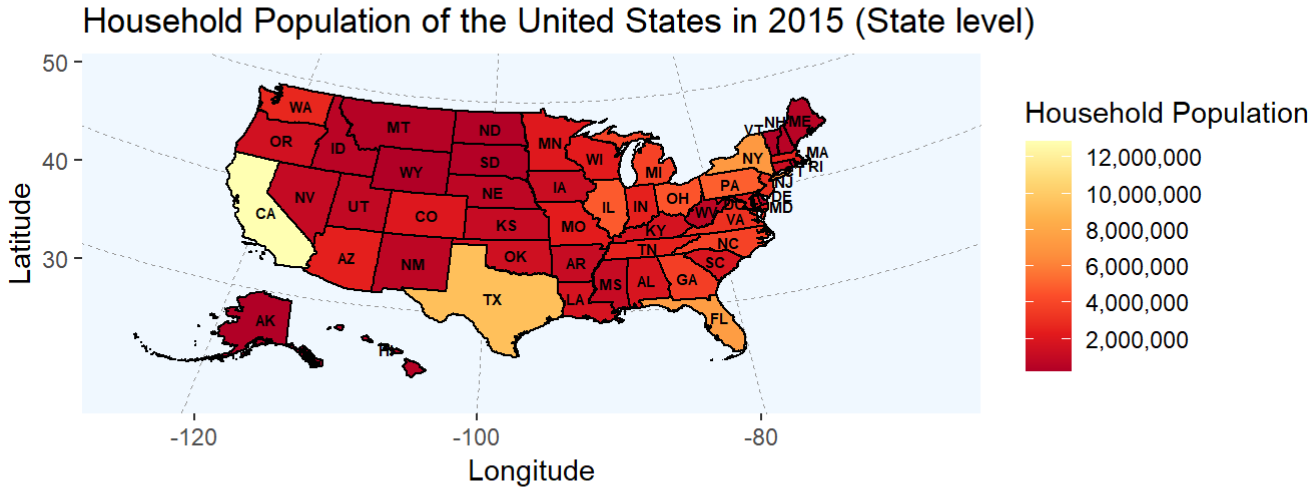



```
# Plotting at county level
ggplot() +
  geom_polygon(data = urbnmapr::counties, mapping = aes(x = long, y = lat, group = group),
                   fill = "grey", color = "black") +
  coord_map(projection = "albers", lat0 = 39, lat1 = 45) + # Used to alter map projection for
a nicer view
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Map of the United States (County level)", subtitle = "Using the \"urbnmapr\" packa
ge") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```

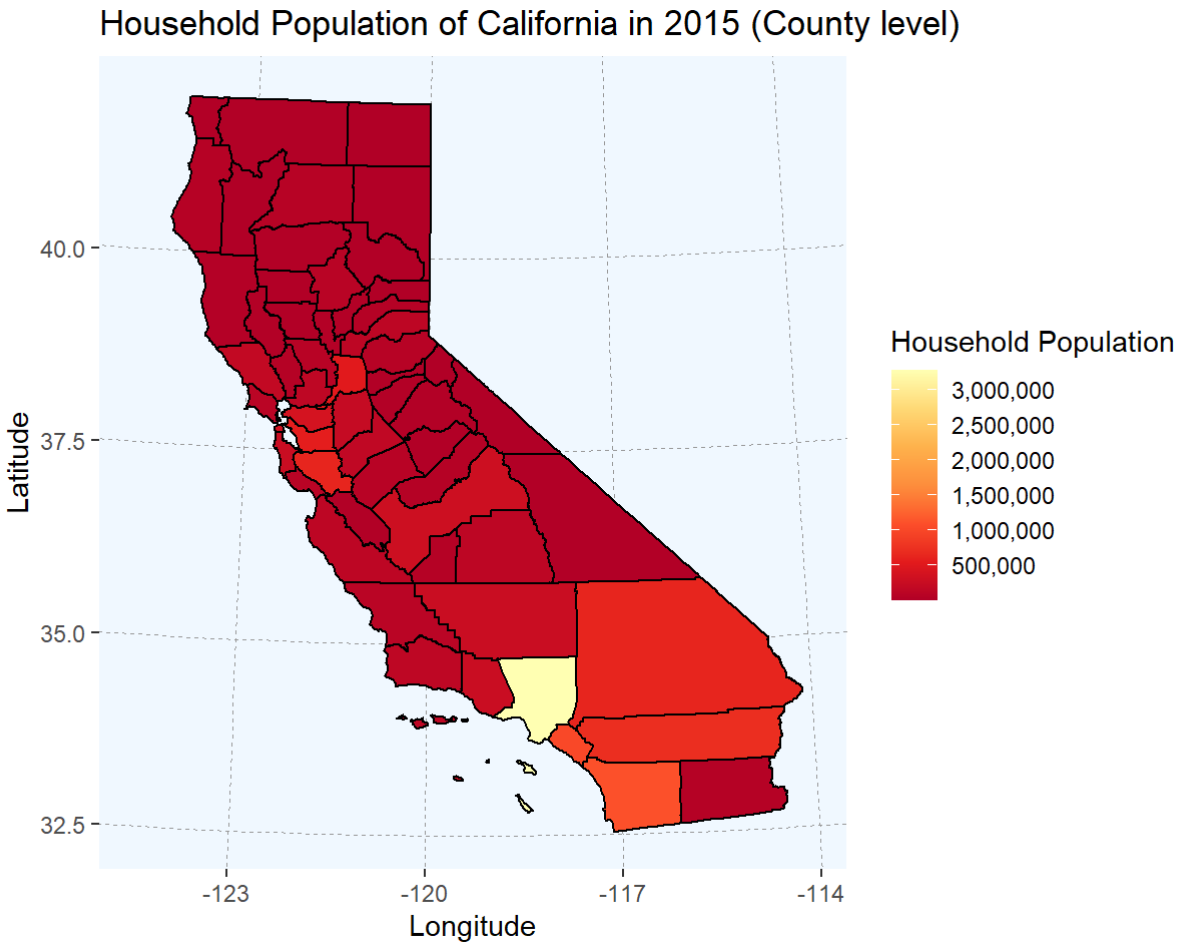


The state dataset (**urbnmapr::statedata**) includes state names, abbreviations, and Federal Information Processing Standards (FIPS) codes — unique numeric identifiers for each county and state. The county dataset (**urbnmapr::countydata**) includes all the state variables plus county names and FIPS codes, making it easy to merge with the data you are working with, whether they include names or codes. Both datasets include sample data from 2015 which you can use to incorporate into your map, such as Household population, Home Ownership Rate, and Median Household Income.

```
# Plotting a map of household populations across states
states_data <- left_join(statedata, urbnmapr::states, by = "state_fips")
states_label_data <- get_urbn_labels(map = "states")

ggplot() +
  geom_polygon(data = states_data, mapping = aes(x = long, y = lat, group = group, fill = hhp
op), color = "black") +
  geom_text(data = states_label_data, aes(label = state_abbv, x = long, y = lat, fontface =
"bold"), size = 2) +
  scale_fill_distiller(name="Household Population",
                       palette = "YlOrRd",
                       breaks = breaks_pretty(n = 5),
                       labels = comma) +
  coord_map(projection = "albers", lat0 = 39, lat1 = 45) + # Used to alter map projection for
a nicer view
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Household Population of the United States in 2015 (State level)") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  theme(aspect.ratio = 0.4)
```




```
# Plotting a map of household populations across counties in California
counties_data <- left_join(countydata, urbnmapr::counties, by = "county_fips")
counties_data <- counties_data[counties_data$state_name == "California",]

ggplot() +
  geom_polygon(data = counties_data, aes(x = long, y = lat, group = group, fill = hhpop), col
or = "black") +
  scale_fill_distiller(name="Household Population",
                       palette = "YlOrRd",
                       breaks = breaks_pretty(n = 5),
                       labels = comma) +
  coord_map(projection = "albers", lat0 = 39, lat1 = 45) + # Used to alter map projection for
a nicer view
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Household Population of California in 2015 (County level)") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```

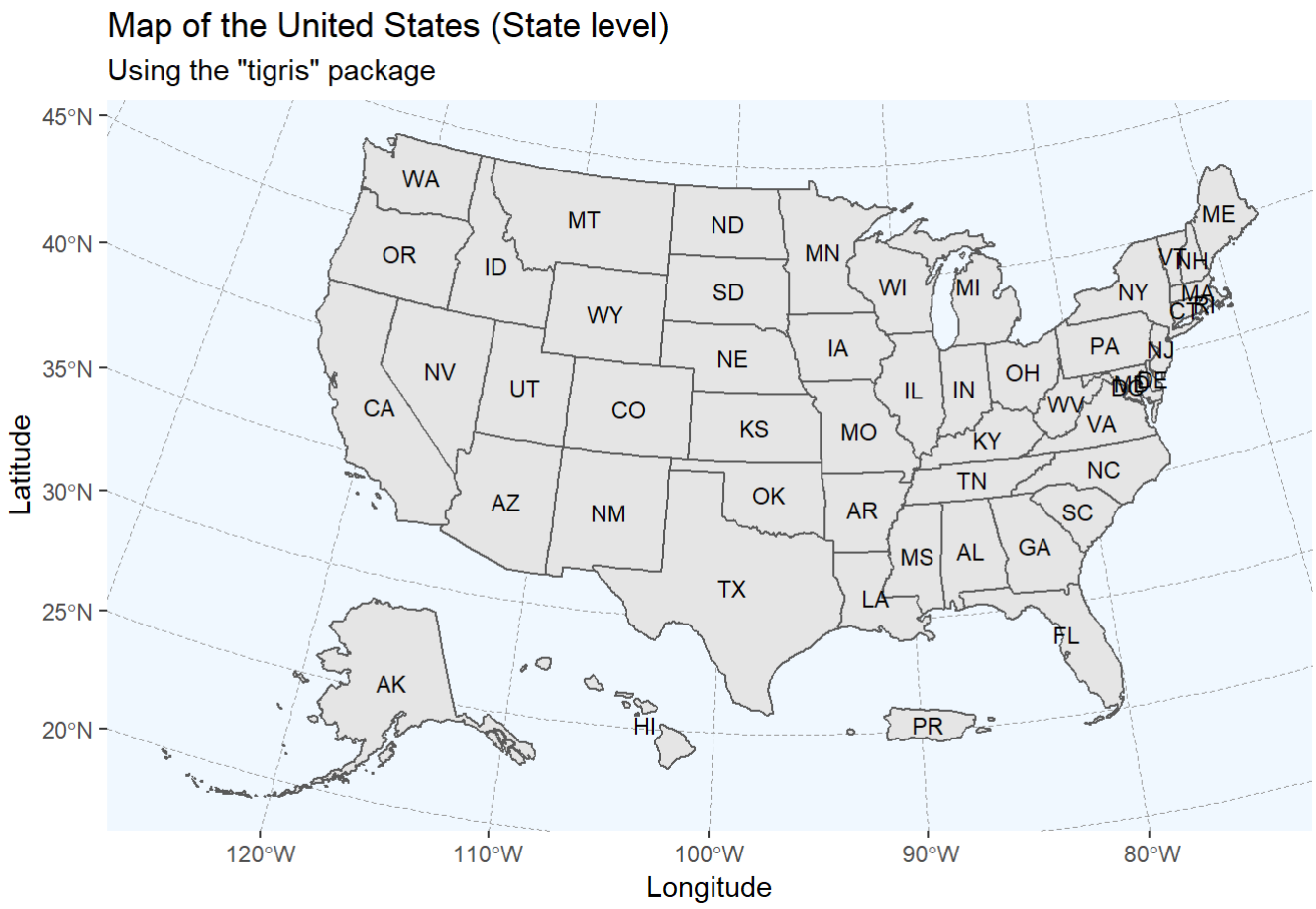


4. Using the **tigris** Package

Aside from the **urbnmapr** package, The **tigris** package can be used to plot spatial data in the United States as well. **tigris** is an R package that allows users to directly download and use TIGER/Line shapefiles (https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html (https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html)) from the US Census Bureau.

This package has several advantages over the **urbanmapr** package:
1. It allows flexibility in plotting maps, such as being able to use “sf” and “sp” classes, being able to rescale maps (Positions of Hawaii, Alaska etc.) to your liking, and being able to plot a greater variety of maps aside from States and Counties (such as Core Based Statistical Areas)
2. It contains many functions, datasets and shapefiles that will aid you in spatial analysis within the United States. A full list of functions and their descriptions can be found here. (https://www.rdocumentation.org/packages/tigris/versions/1.4.1)
3. It is constantly being updated at a more frequent rate, so all information inside are up-to-date.

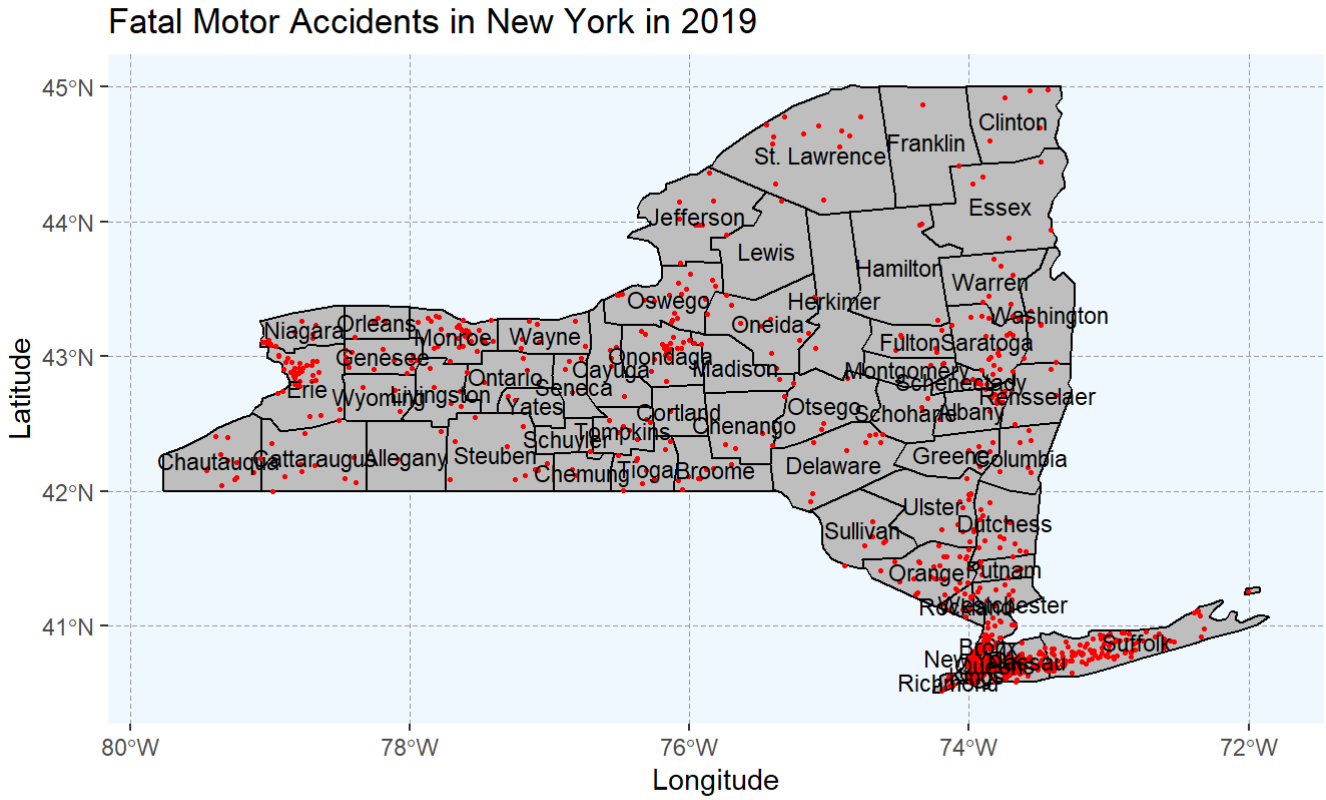
```
library(sf) # To obtain coordinates of centroids to label states

# To plot a map of the United States at State level - similar to the one done with urbnmapr p
ackage
us_states <- states(cb = TRUE, resolution = "20m", progress_bar = FALSE) %>%
  shift_geometry(position = "below") #Additional step to ensure that Hawaii and Alaska are wi
thin the visualisations

# To add in coordinates of centroids of all states into the US spatial dataset
us_states <- cbind(us_states,st_coordinates(st_centroid(us_states$geometry)))

ggplot(data = us_states) +
  geom_sf() +
  geom_text(aes(label = STUSPS, x = X, y = Y), size = 3) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Map of the United States (State level)", subtitle = "Using the \"tigris\" package"
) +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```



Here, we can see that Puerto Rico is included in the dataset provided by **tigris**, but not in **urbanmapr**.

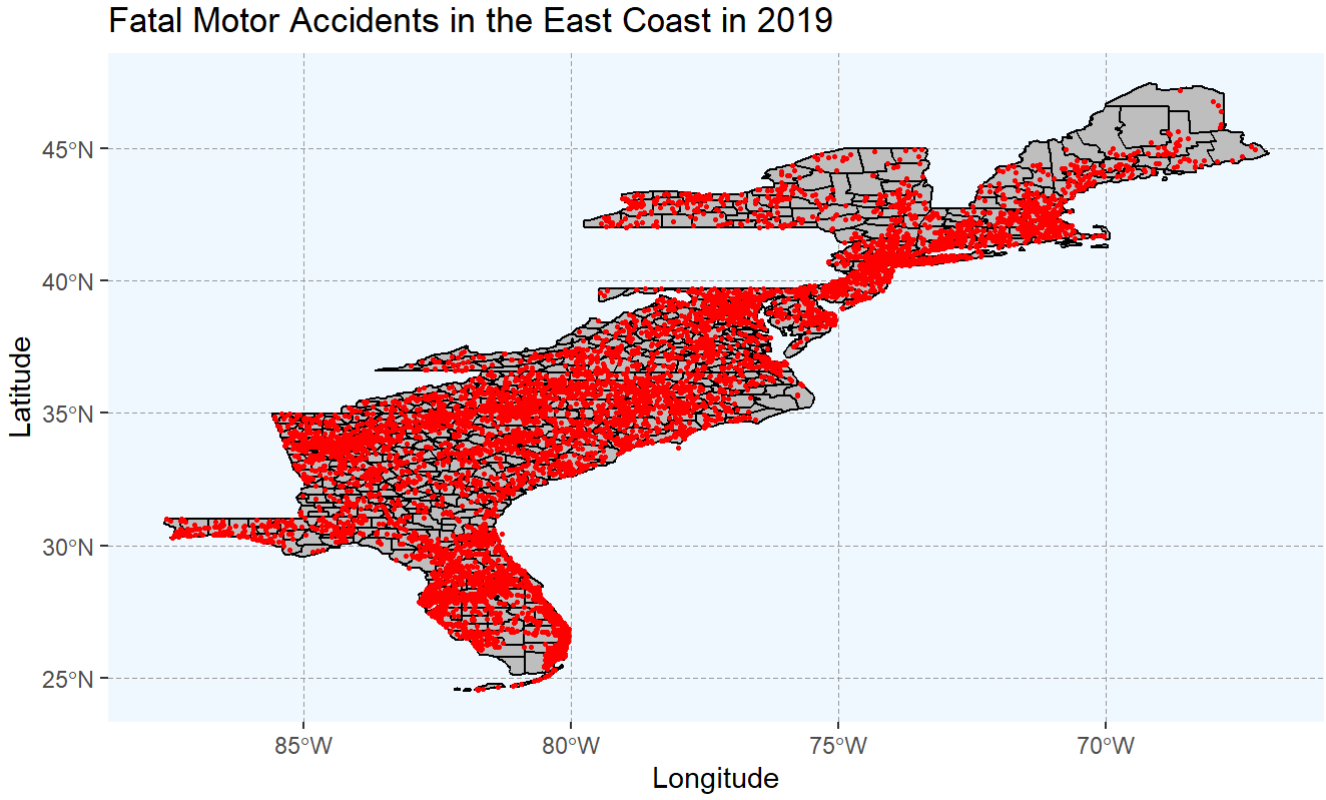
Using the **counties()** function, we can plot a county level map of any state we desire. In this case, we are looking at the locations of all fatal motor accidents in New York that have been reported by the United States Department of Transporation in 2019.

```
ny_data <- counties(state = "NY", cb = TRUE, resolution = "20m", progress_bar = FALSE)
accidents <- read.csv("accident.csv")
# Longitude filter added to remove outliers
accidents <- accidents[accidents$STATENAME == "New York" & accidents$LONGITUD < 0,]
# Adding Centroid coordinates for labelling purposes
ny_data <- cbind(ny_data, st_coordinates(st_centroid(ny_data$geometry)))

ggplot() +
  geom_sf(data = ny_data, fill = "grey", color = "black") +
  geom_point(data = accidents, aes(x = LONGITUD, y = LATITUDE), color = "red", size = 0.5) +
  geom_text(data = ny_data, aes(label = NAME, x = X, y = Y), size = 3) +
  theme(aspect.ratio = 0.55) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Fatal Motor Accidents in New York in 2019") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```



Using the **counties()** function, you can also plot counties in multiple states easily, by entering your desired states in the “state” argument. In this case we are looking at the locations of all fatal motor accidents in the East Coast of the United States.

```
eastcoast <- c("Maine", "New Hampshire", "Massachusetts", "Rhode Island", "Connecticut", "New
York", "New Jersey", "Delaware", "Maryland", "Virginia", "North Carolina", "South Carolina",
"Georgia", "Florida")
eastcoast_data <- counties(state = eastcoast, cb = TRUE, resolution = "20m", progress_bar = F
ALSE)
accidents <- read.csv("accident.csv")
# Longitude filter added to remove outliers
accidents <- accidents[(accidents$STATENAME %in% eastcoast) & accidents$LONGITUD < 0,]

# Plotting the locations of all fatal motor accidents in the east coast area of the United St
ates
ggplot() +
  geom_sf(data = eastcoast_data, fill = "grey", color = "black") +
  geom_point(data = accidents, aes(x = LONGITUD, y = LATITUDE), color = "red", size = 0.5) +
  theme(aspect.ratio = 0.55) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Fatal Motor Accidents in the East Coast in 2019") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```

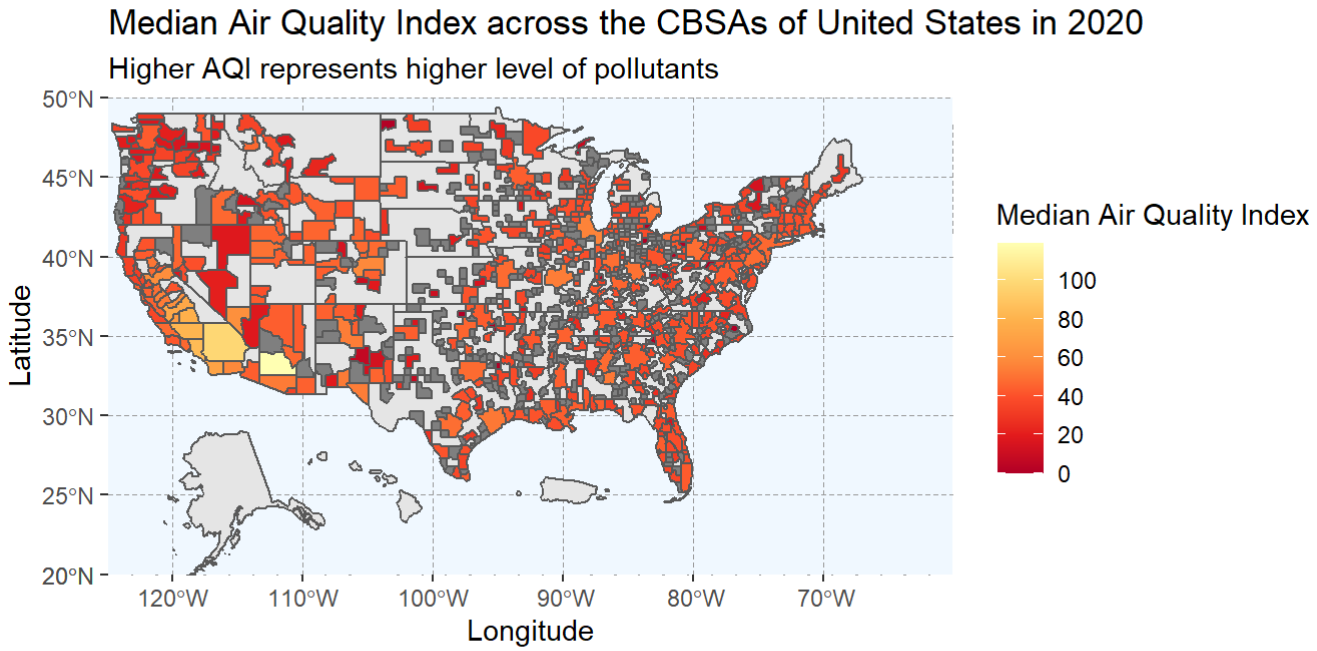


Another benefit of this package is its ability to plot maps of the United States based on its **Core Based Statistical Area (CBSA)**. The US Census Bureau defines these areas as follows: “A metro area contains a core urban area of 50,000 or more population, and a micro area contains an urban core of at least 10,000 (but less than 50,000) population. Each metro or micro area consists of one or more counties and includes the counties containing the core urban area, as well as any adjacent counties that have a high degree of social and economic integration (as measured by commuting to work) with the urban core.”

When data is divided into CBSAs instead of states or counties in the United States, this package will benefit users greatly.

```
# Air quality data obtained from the United States EPA website
aqi <- read.csv("annual_aqi_by_cbsa_2020.csv")

# Obtaining the United States' CBSA spatial data
cbsa <- core_based_statistical_areas(cb = TRUE, resolution = '20m', progress_bar = FALSE)
cbsa$GEOID <- as.numeric(cbsa$GEOID)

cbsa <- left_join(cbsa, aqi, by = (c("GEOID" = "CBSA.Code")))
# plotting the United States Map at Core Based Statistical Area level
ggplot(data = cbsa) +
  geom_sf(data = us_states) + # To maintain outer borders of the United States
  geom_sf(aes(fill = Median.AQI)) +
  # To show majority of areas, otherwise the plot would be extremely zoomed out
  coord_sf(xlim = c(-125, -60), ylim = c(20,50), expand = FALSE) +
  xlab("Longitude") +
  ylab("Latitude") +
  scale_fill_distiller(name="Median Air Quality Index",
                       palette = "YlOrRd",
                       breaks = breaks_pretty(n = 5)) +
  ggtitle("Median Air Quality Index across the CBSAs of United States in 2020",
          subtitle = "Higher AQI represents higher level of pollutants") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue"))
```

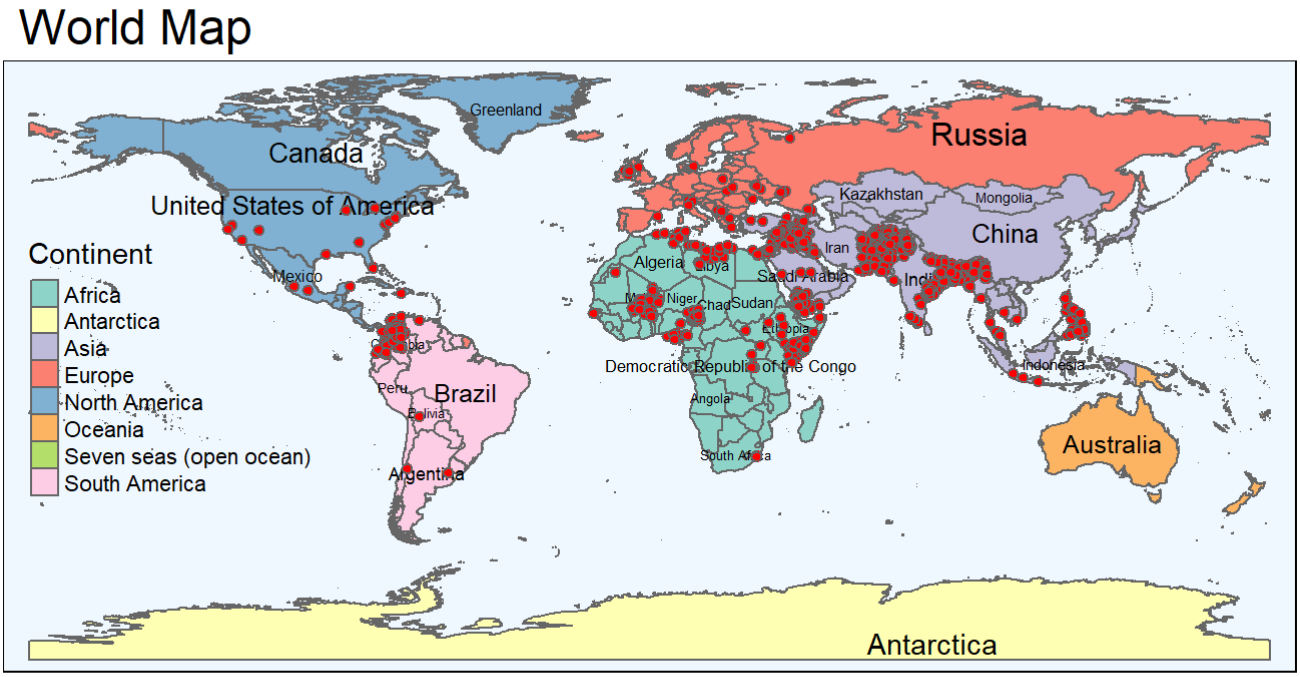
```
## Warning in st_cast.GEOMETRYCOLLECTION(X[[i]], ...): only first part of
## geometrycollection is retained
```



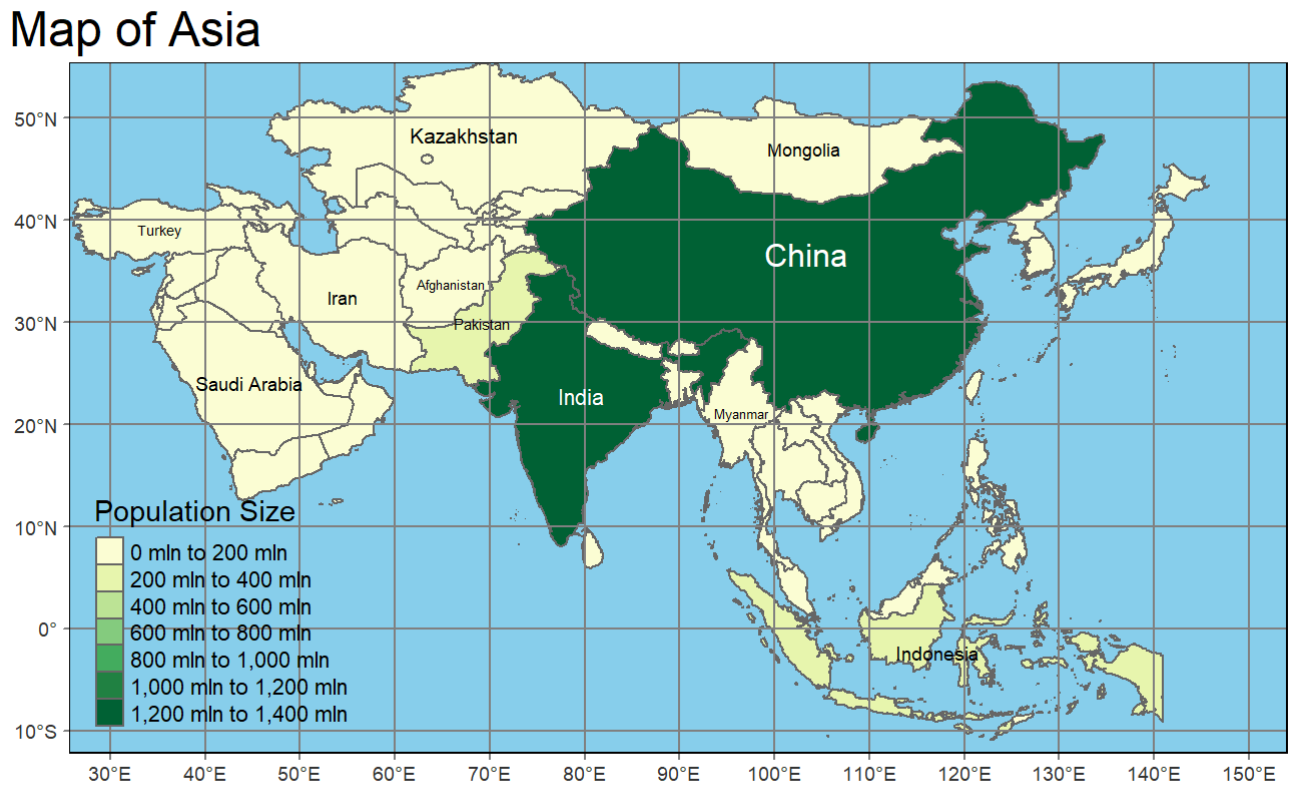
5. Plotting maps using the **tmap** Package

Aside from **ggplot2**, **tmap** is another package you can use to plot spatial data. In fact, this package is specifically tailored for plotting maps. We'll use the examples of bombings around the world, and population of countries in Asia to plot our maps on this package.

```
library(sf)
library(tmap)

world <- ne_countries(scale = "large", returnclass = "sf")

# Same dataset as shown previously
terrorism <- read.csv("globalterrorismdb_0919dist.csv")
terrorism2 <- terrorism[terrorism$attacktype1_txt == "Bombing/Explosion" & terrorism$iyear ==
2018,]

# To convert this dataset to "sf" in order to be compatible with tmaps
terrorism_clean <- drop_na(terrorism2, c("longitude","latitude")) # sf data do not allow miss
ing coordinates
projcrs <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0" # To specify pro
jection (optional)
terrorism_clean <- st_as_sf(terrorism_clean, coords = c("longitude","latitude"), crs = projcr
s)


tm_shape(world) +
  tm_polygons(col = "continent", title = "Continent") + # When variables are categorical, bes
t not to specify palette
  tm_text("admin", size = "AREA") + # Size = "AREA" varies the size of each text based on its
polygon size
  tm_layout(main.title = "World Map", bg.color = "aliceblue", legend.position = c("left","cen
ter")) +
  tm_shape(terrorism_clean) + tm_symbols(size = 0.15, col = "red")
```




```
asia <- ne_countries(scale = "large", continent = "asia", returnclass = "sf")
asia$pop_est <- as.numeric(asia$pop_est)

tm_shape(asia) +
  tm_polygons(col = "pop_est", title = "Population Size", palette = "YlGn") +
  tm_graticules() +
  tm_text("admin", size = "AREA") +
  tm_layout(main.title = "Map of Asia", bg.color = "skyblue", legend.position = c("left","bot
tom"), inner.margins = 0)
```




```
  # tm_layout(main.title = "Map of Asia", bg.color = "skyblue", legend.outside = TRUE, inner.
margins = 0)
  # Use this tm_layout line instead of the one above to move legend outside of the plot
```

Preset map settings are also available using the **tmap_style()** funcion

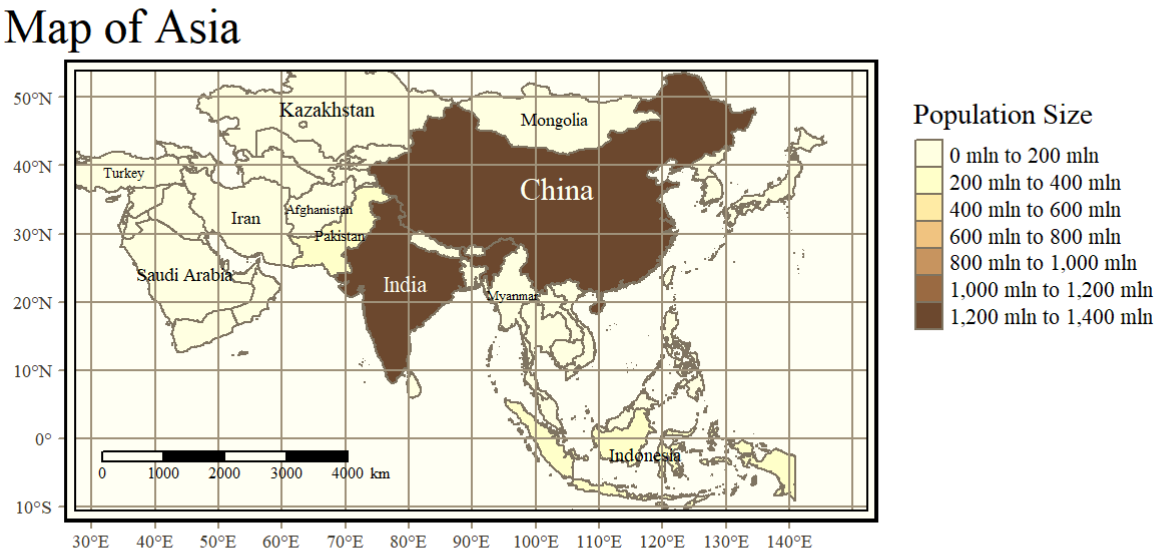
```
tmap_style("classic")
```

```
## tmap style set to "classic"
```

```
## other available styles are: "white", "gray", "natural", "cobalt", "col_blind", "albatros
s", "beaver", "bw", "watercolor"
```

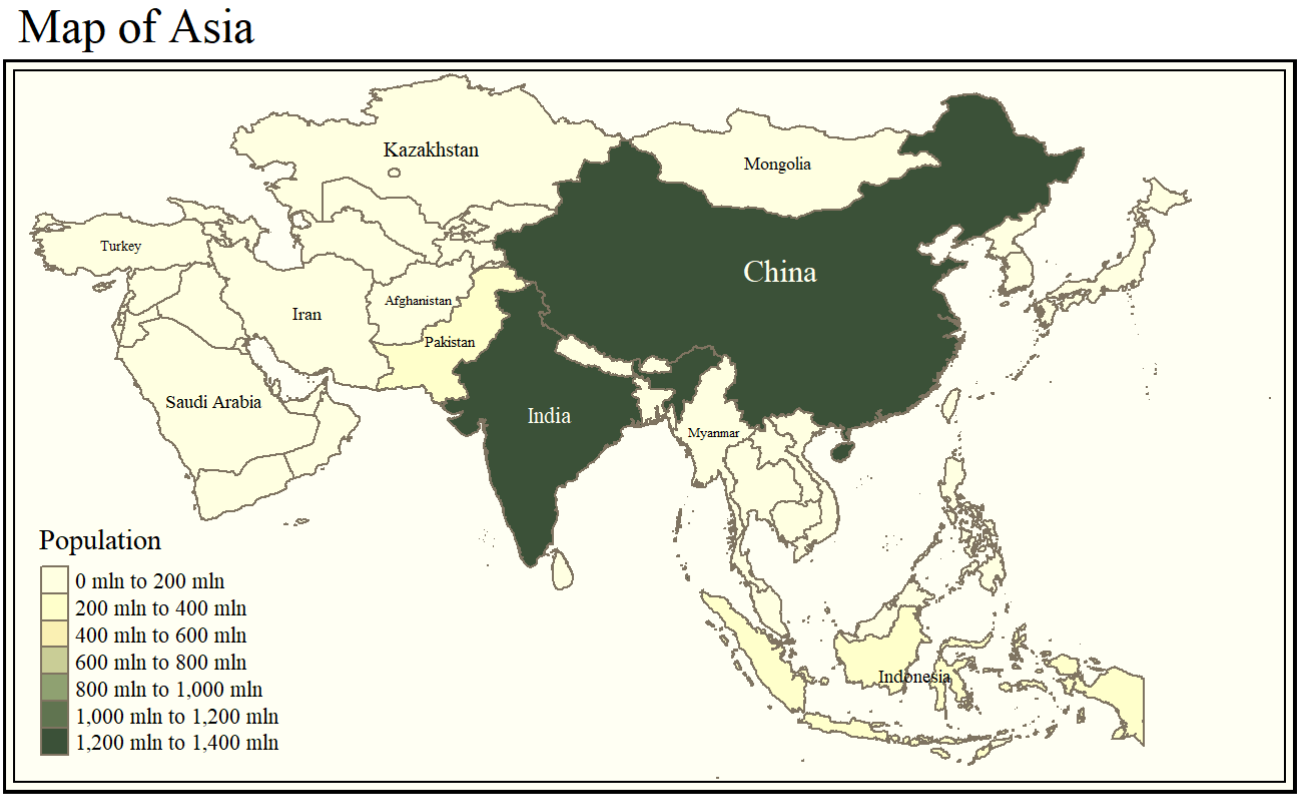
```
# other available styles are: "white", "gray", "natural", "cobalt", "col_blind", "albatross",
"beaver", "bw", "watercolor"

tm_shape(asia) +
  tm_polygons(col = "pop_est", title = "Population Size") +
  tm_graticules() +
  tm_text("admin", size = "AREA") +
  tm_layout(main.title = "Map of Asia", legend.outside = TRUE, inner.margins = 0) +
  tm_scale_bar(position = "left")
```

```
## Scale bar set for latitude km and will be different at the top and bottom of the map.
```



Maps can also be made with just one function, qtm(), which stands for Quick Thematic Map.

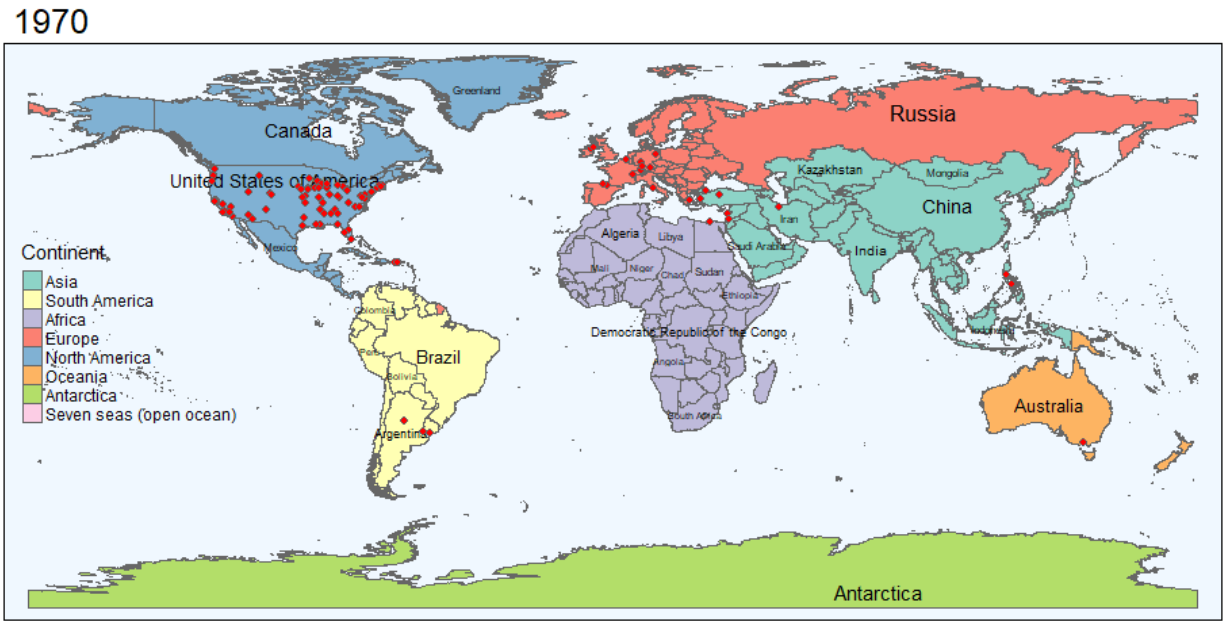
```
qtm(asia, "pop_est", fill.palette = "YlGn", text = "admin", text.size = "AREA", fill.title =
"Population", main.title = "Map of Asia")
```



Using the **tmap** package, we can also create animated maps to replace faceted maps, which might not be ideal as faceted maps squeeze multiple maps into a single screen and might not be easy to read. In the example below, we can use animated maps to show how the distribution of bombings across the world varied from year to year, instead of showing a single year like in the above example. The **gifski** package will also be required when converting the faceted maps into a gif file.

```
library(gifski)

terrorism3 <- terrorism[terrorism$attacktype1_txt == "Bombing/Explosion",]

# To convert this dataset to "sf" in order to be compatible with tmaps
terrorism_clean2 <- drop_na(terrorism3, c("longitude","latitude")) # sf data do not allow mis
sing coordinates
projcrs <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0" # To specify pro
jection (optional)
terrorism_clean2 <- st_as_sf(terrorism_clean2, coords = c("longitude","latitude"), crs = proj
crs)

# Creating the faceted maps
terrorism_anim <- tm_shape(world) +
  tm_polygons(col = "continent", title = "Continent") + # When variables are categorical, bes
t not to specify palette
  tm_text("admin", size = "AREA") + # Size = "AREA" varies the size of each text based on its
polygon size
  tm_layout(bg.color = "aliceblue", legend.position = c("left","center")) +
  tm_shape(terrorism_clean2) + tm_symbols(size = 0.15, col = "red") +
  tm_facets(along = "iyear", free.coords = FALSE)

# Converted the faceted maps into a gif
tmap_animation(terrorism_anim, filename = "terrorism_anim.gif", delay = 25) # Delay in terms
 of 1/100th of a second
```

After running the above code, you'll be obtain a gif file as you can see below:



6. Loading shapefiles downloaded online for spatial visualisation

You can also download your own shapefiles, and load them into r. As an example, when we plotted a map of Singapore, the boundaries provided by packages covered before only covered up to the various major regions (i.e. Central, NorthWest, etc.). However, we can also download Singapore's polygon data (.shp files) from

public data available. An example can be found here. (https://data.gov.sg/dataset/master-plan-2014-planning-area-boundary-web?resource_id=2ab23cb2-b1a4-4b1a-a9e1-b9cad0ac159b) Afterwards, we can load the file in to r using the **readOGR()** function as follows:

```
sg_plan <- readOGR(dsn = "~/RAStuff", layer = "MP14_PLNG_AREA_WEB_PL")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:\Users\kerwi\OneDrive\Documents\RAStuff", layer: "MP14_PLNG_AREA_WEB_PL"
## with 55 features
## It has 12 fields
```

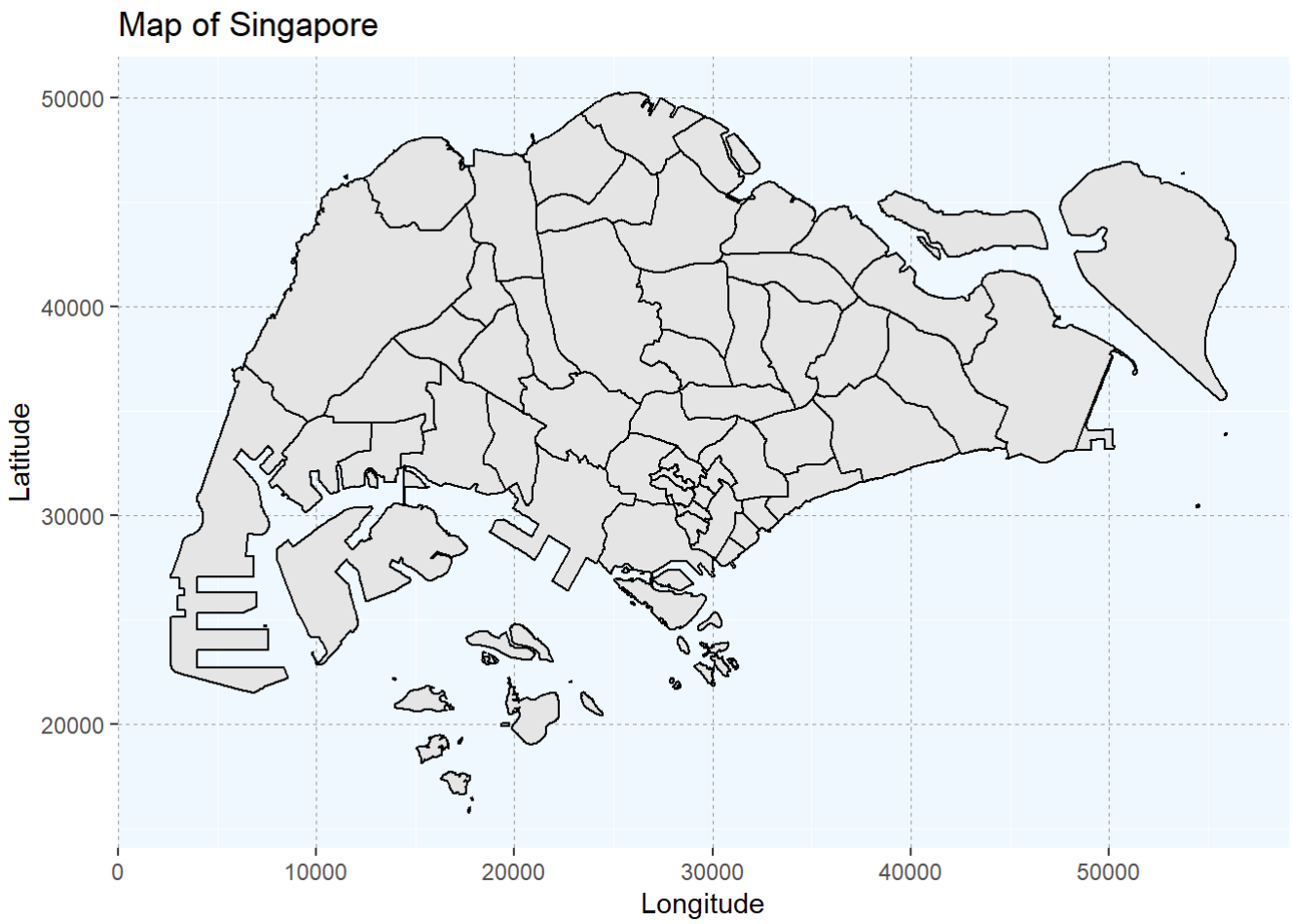
```
# dsn refers to the path of your folder containing your .shp file, layer refers to the name o
f the .shp file without the .shp extension.

summary(sg_plan)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##         min      max
## x  2667.538 56396.44
## y 15748.721 50256.33
## Is projected: TRUE
## proj4string :
## [+proj=tmerc +lat_0=1.36666666666667 +lon_0=103.833333333333 +k=1
## +x_0=28001.642 +y_0=38744.572 +datum=WGS84 +units=m +no_defs]
## Data attributes:
##     OBJECTID     PLN_AREA_N         PLN_AREA_C           CA_IND
##  Min.   : 1.0   Length:55          Length:55          Length:55
##  1st Qu.:14.5   Class :character   Class :character   Class :character
##  Median :28.0   Mode  :character   Mode  :character   Mode  :character
##  Mean   :28.0
##  3rd Qu.:41.5
##  Max.   :55.0
##    REGION_N           REGION_C           INC_CRC           FMEL_UPD_D
##  Length:55          Length:55          Length:55          Length:55
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##      X_ADDR          Y_ADDR        SHAPE_Leng       SHAPE_Area
##  Min.   : 5688   Min.   :25070   Min.   :  4386   Min.   :  830273
##  1st Qu.:21930   1st Qu.:31596   1st Qu.: 13238   1st Qu.: 6630102
##  Median :28530   Median :34689   Median : 16933   Median :10100544
##  Mean   :27627   Mean   :36004   Mean   : 20852   Mean   :14217178
##  3rd Qu.:32092   3rd Qu.:40148   3rd Qu.: 22022   3rd Qu.:16652231
##  Max.   :50425   Max.   :48595   Max.   :111547   Max.   :69748299
```

Afterwards, we can look to plot the polygon data obtained.

```
# To plot the polygon data downloaded
ggplot() +
  geom_polygon(data = sg_plan, aes(x = long, y = lat, group = group), color = "black", fill =
"gray90") +
  theme(panel.grid.major = element_line(color = "gray60", linetype = "dashed", size = 0.25),
        panel.background = element_rect(fill = "aliceblue")) +
  xlab("Longitude") +
  ylab("Latitude") +
  ggtitle("Map of Singapore")
```

```
## Regions defined for each Polygons
```




```
# Alternative method to plot
tm_shape(sg_plan) + tm_polygons() + tm_text("PLN_AREA_N", size = "AREA")
```

```
## Warning in sp::proj4string(obj): CRS object has comment, which is lost in output
```

Compared to the previous maps of Singapore shown, this is more detailed when it comes to the number of border lines surrounding the various regions of Singapore. Now we are able to view more specific areas such as “Yishun”, “Tuas” and so on.

It is important to note that a shapefile should consist of at least four files: .shp (the geometry), .dbf (the attributes), .shx (the index that links the two, and .prj (the coordinate reference system). If the .prj file is missing, a warning is given. If any other file is missing an error occurs (although one could in principle recover the .shx from the .shp file).

When downloading the polygon data file provided in the link above, it is important to copy all the files in the folder (instead of just the .shp file), or the **readOGR()** function will not work.

Shiny App Development

Dr. Liu Qizhang

15 March 2019

Introduction

This file covers the basics of Shiny app development.

Overview

The R Shiny package has a simple yet powerful architecture, consisting of just two components, the User Interface (UI) and Server.

The UI is what users can see and interact with. Anything that the users do through the UI will be passed on as input to the Server.

The Server takes input from the UI and executes predefined logic as a response. The result is then passed back to the UI as output.

The UI then displays the output to users.

It is possible to run both the UI and Server on your local computer as a **standalone app** or host it on a **cloud server** so that it is available on the internet.

Advantages

One advantage of Shiny is that code can be converted into HTML script, allowing us to directly embed it in an HTML page. For example, you could create your own e-commerce site with an R shiny app embedded in it.

First Shiny App

We will build our first Shiny app to show Sentosa Island on the Singapore map. Our first leaflet map was also about Sentosa Island, and it provided a zoom in and zoom out button for us to control the map.

Now, we want our Shiny app to control the zoom level with a slider. The slider should have a default zoom level and when we slide the bar, the map will zoom accordingly.

Firstly, we will want to create a new R script in RStudio and save it as ‘app.R’ - the default name recognized by R as a Shiny app. It is important to use this name precisely. Otherwise, R will not know you are developing an app.

For **windows**, you can use the shortcut **CTRL + SHIFT + N** to create a new R script, then use the shortcut **CTRL + S** to save it as ‘app.R’.

For **mac**, you can use the shortcut **COMMAND + SHIFT + N** to create a new R script, then use the shortcut **COMMAND + S** to save it as ‘app.R’.

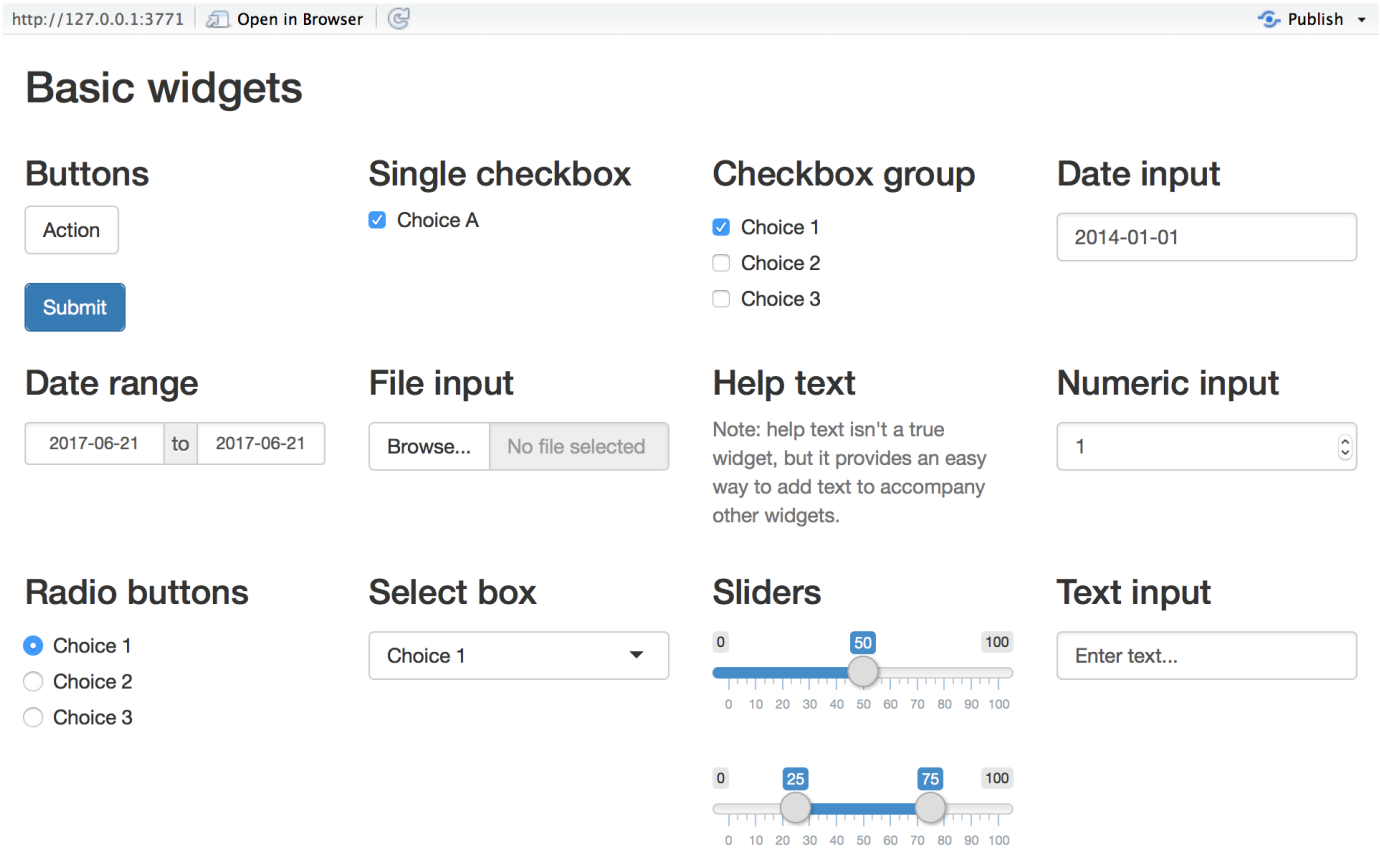
Let's first make use of the default template Shiny code to start. This consists of three parts: **UI**, **Server** and **Shiny app**.

Copy and paste the following code into your 'app.R' file.

```
library(shiny)

#Define UI for app
ui <- fluidPage()

#Define server logic
server <- function(input, output){}

shinyApp(ui=ui, server=server)
```

A typical UI consists of two parts: **controls** and **outputs**.

Controls are visual elements that allow users to interact with the system. For our first app, the slider is a control.



Notice the sliders section of the image, our goal is to implement that. Let's add the slider to the UI now.

```
ui <- fluidPage(
  sliderInput(inputId = "zoomlevel",
              label = "Map Zooming Level",
              value = 11,
              min = 1,
              max = 20)
)
```

This is done by inserting a line of code into UI. If we run our app.r file (you should notice it now has a 'run app' button where 'run' used to be in a normal R script), you should see

The title of the slider should be the label we set in our code. By default, the slider marker is set at 11, which is our value in the code. Additionally, the range of the slider is set by our min and max values.

Take note of the inputId, it is a unique identifier of the slider and its corresponding values. If you have tested sliding the bar, nothing will happen, because we haven't defined the **outputs**.

UI Outputs

Functions for creating user interface elements that, in conjunction with rendering functions, display different kinds of output from your application.

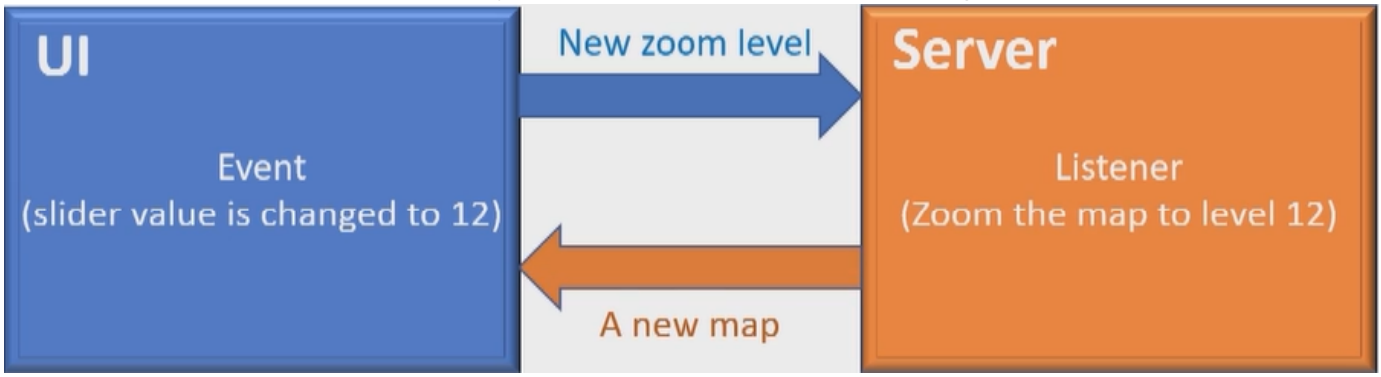
`htmlOutput (uiOutput)`	Create an HTML output element
`plotOutput (imageOutput)`	Create an plot or image output element
`outputOptions`	Set options for an output object.
`tableOutput (dataTableOutput)`	Create a table output element
`textOutput`	Create a text output element
`verbatimTextOutput`	Create a verbatim text output element
`downloadButton (downloadLink)`	Create a download button or link
`Progress`	Reporting progress (object-oriented API)
`withProgress (setProgress, incProgress)`	Reporting progress (functional API)
`modalDialog`	Create a modal dialog UI
`urlModal`	Generate a modal dialog that displays a URL
`showModal (removeModal)`	Show or remove a modal dialog
`showNotification (removeNotification)`	Show or remove a notification

Shiny supports many different types of outputs as shown above, including, table, text, image, plot, map, etc. As we are going to show leaflet map, we will use *leafletOutput()*.

Our ui should now look like this:

```
ui <- fluidPage(
  sliderInput(inputId = "zoomlevel",
              label = "Map Zooming Level",
              value = 11,
              min = 1,
              max = 20),  #Notice the added comma here
  leafletOutput(outputId = "plotMap")
)
```

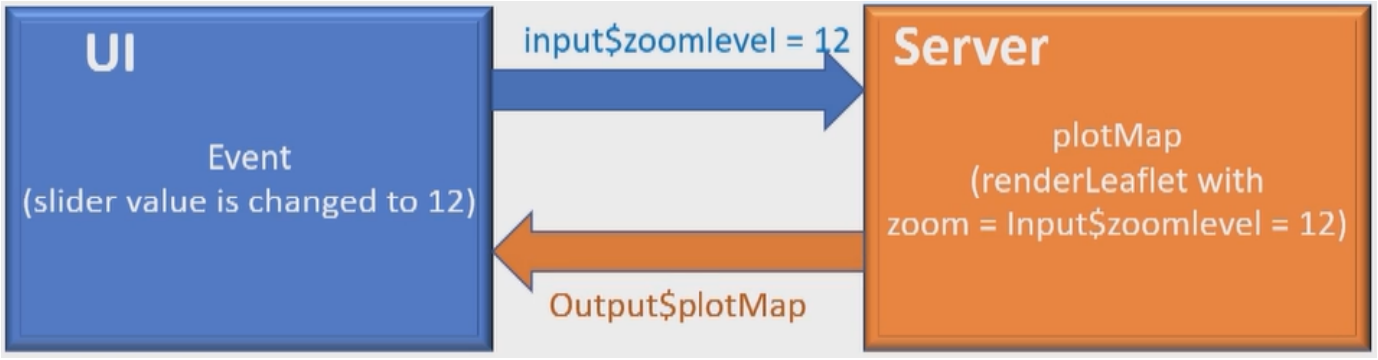
We also need to give *leafletOutput()* a unique outputId. Whenever we make changes to a UI control, an event is triggered. The event passes the corresponding changes to the server, which has a listener reacting to the event to produce we would expect to say. In this case a new map with a changed zoom level.



How the listener responses to events are coded in the server function. Let's add the following code to it.

```
server <- function(input, output){
  output$plotMap <- renderLeaflet(
    {
      leaflet() %>% setView(lng = 103.835381, lat = 1.239660,
                            zoom = input$zoomlevel) %>%
        addTiles() %>%
        addMarkers(lat = 1.239660, lng = 103.835381,
                   popup = "Sentosa Cove")
    }
  )
}
```

It defines what the output with id "plotMap" will look like. The output is generated by rendering a leaflet map. The code inside renderLeaflet is exactly the same as the previous lesson, except that zoom level is now set by the input with id "zoomlevel".



Let's recap on how the app works. When the slider value changes to the number 12, input zoom level ("input$zoomlevel") is set to 12. The Server function will render a leaflet with zoom level equal to the input zoom level ("input$zoomlevel"). The result is an output plot map ("output$plotMap") that will be sent back to UI

and displayed there.

So far, we write everything is in ‘app.R’. However, it is possible to split it into two files, ‘ui.R’ and ‘server.R’ as above. These two files together will have the same effect as ‘app.R’.

Again, it is important to **save their file names exactly**.

In summary, to develop a Shiny app. We simply need to define two things. Define a UI control, and define an output model that links to the UI input. The most difficult part is the codes for rendering the desired output, which we have already covered previously.

Shiny App Layout

The core of a Shiny app is the server code, which determines what kind of services your app could provide. However, this doesn’t mean UI is not important.

Recently, User Experience (UX) has gained unprecedented attention in the software industry. Easy of use and a great looking and feeling UI is essential in acquiring and retaining users.

We can make use of the layout template offered in Shiny for a better UI structure, and look and feel.

Think back to the case about health insurance, we saw how the willingness of buying health insurance differed by marital status of customers. What if we wanted to see the impact of the other attributes of the customers, such as sex, housing type, etc on their choices of buying health insurance?

We may want to design an app where we can select different radio buttons for changing bar positioning and a drop-down menu to select a customer attribute.

We will use a **layout** called sidebar layout. It is one of the most popular layouts where controls are organized in a sidebar and a large main space is reserved for output display.

```
ui <- pageWithSidebar(
  headerPanel(),

  sidebarPanel(),

  mainPanel()
)
```

It consists of three components, “headerPanel”, “sidebarPanel” and “mainPanel”, separated by commas.

Header Panel is for us to give the title of the app.

```
ui <- pageWithSidebar(
  headerPanel("How are customers' choices on buying health
              insurance affected?"),

  sidebarPanel(),

  mainPanel()
)
```

Sidebar panel stores all the input controls. In this case, we provide two controls, “selectInput” and “radioButtons”. Each control must have a unique input ID, label and a list of choices.

```
ui <- pageWithSidebar(
  headerPanel("How are customers' choices on buying health
              insurance affected?"),

  sidebarPanel(
    selectInput(inputId = "attribute",
                label = "Customer Attribute",
                choices = c("housing.type", "sex",
                            "is.employed", "marital.stat")),
    radioButtons(inputId = "position",
                 label = "Bar Positioning",
                 choices = c("stack", "dodge", "fill"))
  ),

  mainPanel()
)
```

Main panel simply produces an output with output id “custPlot”.

```
ui <- pageWithSidebar(
  headerPanel("How are customers' choices on buying health
              insurance affected?"),

  sidebarPanel(
    selectInput(inputId = "attribute",
                label = "Customer Attribute",
                choices = c("housing.type", "sex",
                            "is.employed", "marital.stat")),
    radioButtons(inputId = "position",
                 label = "Bar Positioning",
                 choices = c("stack", "dodge", "fill"))
  ),

  mainPanel(
    plotOutput("custPlot")
  )
)
```

Note that we are going to plot a chart using ggplot2, therefore “plotOutput” function is used here.

Server side, we just need to render a ggplot with an output id linked to the outputId defined in UI.

```
ui <- pageWithSidebar(
  headerPanel("How are customers' choices on buying health
              insurance affected?"),

  sidebarPanel(
    selectInput(inputId = "attribute",
                label = "Customer Attribute",
                choices = c("housing.type", "sex",
                            "is.employed", "marital.stat")),
    radioButtons(inputId = "position",
                 label = "Bar Positioning",
                 choices = c("stack", "dodge", "fill"))
  ),

  mainPanel(
    plotOutput("custPlot")
  )
)

server <- function(input, output){
  output$custPlot <- renderPlot(
    {
      ggplot(custdata) +
        geom_bar(aes_string(x=input$attribute, fill="health.ins"),
                 position=input$position)
    }
  )
}
```

Notice that we are using aes_string instead of aes here, an input attribute (“input$attribute”) is passed in as a string and aes will not recognize it as a column of the data frame.

To summarize, adopting a layout is just about putting various parts of Shiny app codes in different components of a given layout template. Shiny provides several other layout templates (https://shiny.rstudio.com/articles/layout-guide.html)

Deploy app on Cloud

An exciting feature of the Shiny app is that you can easily deploy your app to a cloud server and share it with the world. Visit this website (www.shinyapps.io). Sign up an account.

Once signed up, the website will show a step-by-step guide on how to deploy your app.

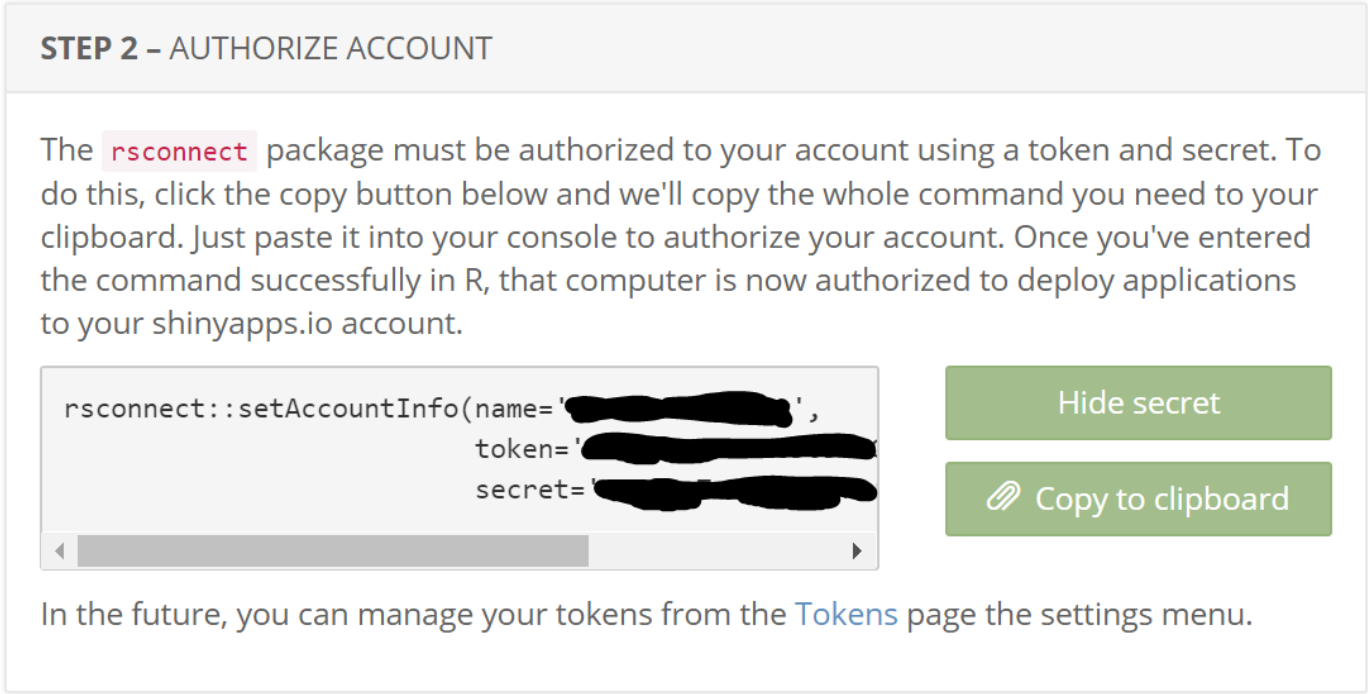
First, install the package ‘rsconnect’. Then, load this package in RStudio.

```
install.packages('rsconnect')
library(rsconnect)
```

Next, you need to authorize the rsconnect package with your shinyapps account.

```
rsconnect::setAccountInfo(name='YOURACCOUNTNAMEHERE',
                          token='YOURPERSONALAPIKEYHERE',
                          secret='YOURSECRETKEYHERE')
```

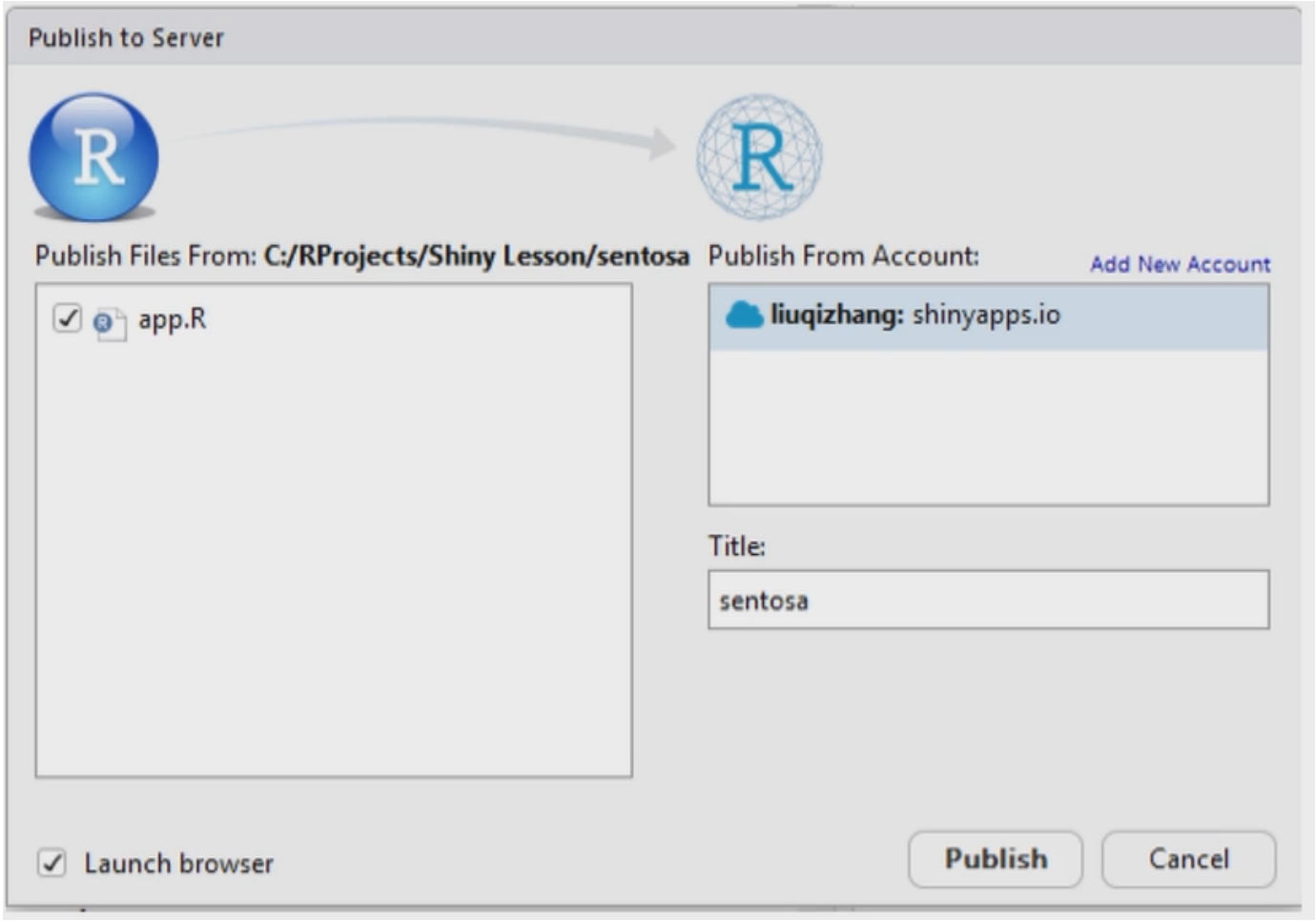
Shinyapps.io has nicely provided a way to copy all this to your clipboard painlessly.



Remember to press the 'Show secret' button, before copying the code to your clipboard. Then run the command in your RStudio. As mentioned in the image above, once you log out, you can manage your tokens from Account > Tokens in the website (www.shinyapps.io).

Finally, to deploy your app, go to your RStudio and click on the button to the right of "Run App". You will see a menu called "Publish Application". Click on it.

In the popup screen:



You must save all the files needed in your app, such as image, data, text, etc. into one single folder, together with “app.R”. In the popup screen, select all the necessary files and then give a proper title to your app.

Then, click “Publish” to deploy it to the cloud server. Once the deployment is completed, a website will pop up in your web browser, showing your app. Record that URL in order to share it with others. You can also manage your apps in the website (www.shinyapps.io).