Base `R` and `tidyverse` refresher

Baris Soybilgen

Contents

Introduction 1

Data structures in `R` 2
- `vector` 2
- `matrix` 4
- `list` 5
- `data frame` 7
- `factor` 8

Control Statements 9
- `if-else` 9
- Loops 10

Functions in `R` 11

`apply()` family in `R` 13

`tidyverse` 14
- `dplyr` 14
- `tidyr` 17
- Relational Data with `dplyr` 19
- `lubridate` 21
- `purrr` 22
- `ggplot2` 24
- Modeling with `tidyverse` 30

Introduction

This refresher shows fundamentals of `R` programming language for data analysis in business and economics. I include both base `R` and `Tidyverse` environment as `Tidyverse` is the mainstream R platform for conducting data science. I design this refresher for my undergraduate economics and business students to follow my `R` based machine learning and data analysis classes better. Most of the examples and definitions are derived from Matloff's (2011) "The Art of `R` Programming" and Wickham & Grolemund's (2017) "`R` for Data Science: Import, Tidy, Transform, Visualize, and Model Data". Therefore, I also encourage you to read those books whenever you don't understand the examples. There are also some examples from DataCamp's Introduction to R and Intermediate R Classes. I always give my students 6 months academic access to my students. It is a good start usually for business students who don't have any programming background. I also add some my examples related with COVID 19 and social sciences. Future Work: More examples will be added instead of examples from Wickham & Grolemund (2017). Animated graphs will be added. `R Shiny` will be added.

Data structures in R

vector

`Vector` is the basic data structure of the `R` programming language. We can access values (index) in vectors using `[]`.

```
x <- c(5,7,9)
x
## [1] 5 7 9
x[3]
## [1] 9
x[2:3]
## [1] 7 9
```

We can apply functions over `vectors`.

```
length(x)
## [1] 3
mode(x) # get or set the type or storage mode of an object.
## [1] "numeric"
typeof(x) # determines the (R internal) type or storage mode of any object
## [1] "double"
class(x) # displays the class of an object
## [1] "numeric"
```

`Vector` can contain a single data type of logical, integer, double, or character.

```
y <- T
str(y)
##  logi TRUE
z <- c(T,21)
str(z)
##  num [1:2] 1 21
v <- c(T,21,"abc")
str(v)
##  chr [1:3] "TRUE" "21" "abc"
```

We can manipulate `vectors` easily using indexing techniques.

```
x <- c(5,7,9,11)
x <- c(x[1:3],999,x[4])
str(x)
##  num [1:5] 5 7 9 999 11
str(x[c(1,3)])
##  num [1:2] 5 9
str(x[-1]) # minus is for deleting elements
##  num [1:4] 7 9 999 11
str(x[-1:-2])
##  num [1:3] 9 999 11
str(x[1:(length(x)-1)])
##  num [1:4] 5 7 9 999
str(x[-length(x)])
##  num [1:4] 5 7 9 999
```

We can further generate `vectors` with `seq()`, `rep()`, and `vector()` functions.

```
str(seq(from = 9,to = 27,by = 3))
##  num [1:7] 9 12 15 18 21 24 27
str(seq(from = 1,to = 5,length = 10))
##  num [1:10] 1 1.44 1.89 2.33 2.78 ...
str(rep(NA,4))
##  logi [1:4] NA NA NA NA
str(rep(c(5,7,9),3))
##  num [1:9] 5 7 9 5 7 9 5 7 9
str(vector(length = 2))
##  logi [1:2] FALSE FALSE
```

One of the important features of `vectors` in `R` is recycling. Applying an operation to two `vectors` requires them to be the same length, `R` automatically recycles, or repeats, the shorter one, until it is long enough to match the longer one. However, using complex recycling operations is generally a bad idea. Furthermore, `R` will show a warning when you are performing an operation on `vectors`, and the `vectors` are not of the same length.

```
c(5,7,9) + c(21,23,25,27,29)
## Warning in c(5, 7, 9) + c(21, 23, 25, 27, 29): longer object length is not a
## multiple of shorter object length
## [1] 26 30 34 32 36
```

For advanced indexing, we can use `comparison operators` and `subset()`. We can also use `which()` to get the actual index value.

```
z <- c(3,5,7,8)
str(z[z*z > 12])
##  num [1:3] 5 7 8
z[z > 5] <- 0
str(z)
##  num [1:4] 3 5 0 0
str(subset(z,z > 1))
##  num [1:2] 3 5
which(z*z > 1)
## [1] 1 2
```

To check whether all elements in a `vector` are true according to the given argument, we use `all()`. To check whether at least one of the values in a `vector` is true according to the given argument, we use `any()`.

```
x <- 1:10
str(any(x > 8))
##  logi TRUE
str(all(x > 3))
##  logi FALSE
```

Warning: Above, we use `NA` to create a `vector` with missing values. The `R` documentation defines `NA` as a logical constant of length one which contains a missing value indicator. In addition to `NA`, `R` has a similar structure called `NULL` which represents the null object and is often returned by expressions and functions whose values are undefined. The important difference between `NA` and `NULL`: `NA` shows missing values and has a logical value, whereas `NULL` values are counted as nonexistent and have no logical value.

```
u <- NULL
length(u)
## [1] 0
NULL > 0
## logical(0)
v <- NA
```

```r
length(v)
## [1] 1
NA > 0
## [1] NA
```

matrix

`Matrix` is a two-dimensional data structure in R programming language. `Matrix` is similar to `vector` but additionally contains the dimension attribute.

```r
m <- rbind(c(10,11),c(12,13))
dim(m)
## [1] 2 2
m
##      [,1] [,2]
## [1,]   10   11
## [2,]   12   13
str(m)
##  num [1:2, 1:2] 10 12 11 13
str(m[2,2])
##  num 13
str(m[1,]) # row 1
##  num [1:2] 10 11
str(m[,2]) # column 2
##  num [1:2] 11 13
```

You can see the difference between `mode()` and `class()` in `matrix`.

```r
mode(m) # get or set the type or storage mode of an object.
## [1] "numeric"
typeof(m) # determines the (R internal) type or storage mode of any object
## [1] "double"
class(m) # displays the class of an object
## [1] "matrix" "array"
```

We can use arithmetic operations over `matrix` and `vector` objects. We can also use matrix multiplication.

```r
x <- c(1,2)
str(m*2)
##  num [1:2, 1:2] 20 24 22 26
str(x*2)
##  num [1:2] 2 4
str(m*x)
##  num [1:2, 1:2] 10 24 11 26
str(m*c(2,4))
##  num [1:2, 1:2] 20 48 22 52
str(m*c(2,4,6,8))
##  num [1:2, 1:2] 20 48 66 104
str(m%*%c(1,1)) # matrix multiplication
##  num [1:2, 1] 21 25
```

`Matrix` can be created by `rbind()`, `cbind()`, and `matrix()`.

```r
matrix(c(1,99,5,7),nrow = 2,ncol = 2)
##      [,1] [,2]
## [1,]    1    5
```

```r
## [2,]   99    7
rbind(c(1,99),c(5,7))
##      [,1] [,2]
## [1,]    1   99
## [2,]    5    7
cbind(c(1,99),c(5,7))
##      [,1] [,2]
## [1,]    1    5
## [2,]   99    7
```

Filtering in `matrices` also conducted similar to `vectors`.

```r
x <- cbind(c(1,5,9),c(2,6,10))
x
##      [,1] [,2]
## [1,]    1    2
## [2,]    5    6
## [3,]    9   10
str(x[x[,2] >= 3,])
##  num [1:2, 1:2] 5 9 6 10
z <- c(5,10,15)
str(x[z %% 2 == 1,])
##  num [1:2, 1:2] 1 9 2 10
str(which(x > 2))
##  int [1:4] 2 3 5 6
```

Warning: When conducting filtering and indexing on a `matrix`, if one dimension is reduced to 1, `R` could convert a `matrix` to a `vector`. To prevent this, we use `drop=F`.

```r
x <- cbind(c(1,5,9),c(2,6,10))
str(x) # matrix
##  num [1:3, 1:2] 1 5 9 2 6 10
str(x[1,]) # vector
##  num [1:2] 1 2
str(x[1,,drop = F]) # matrix
##  num [1, 1:2] 1 2
```

In `R`, a `matrix` with more than 2 dimensions is called an `array`. We can create an `array` using `array()`.

```r
a <- array(data = c(1,2,3,4,5,6,7,8),dim = c(2,2,2))
str(a)
##  num [1:2, 1:2, 1:2] 1 2 3 4 5 6 7 8
str(attributes(a)) # return dimensions of the array as list
## List of 1
##  $ dim: int [1:3] 2 2 2
```

`list`

`List` is the object which contains elements of different types – like strings, numbers, `vectors`, and another `list` inside it. `List` can also contain a `matrix` or a function as its elements. We can create a `list` using `list()`.

```r
j <- list(name = c("Jane", "Mary"), grade = c(60, 82), foreign = T)
str(j)
## List of 3
##  $ name   : chr [1:2] "Jane" "Mary"
```

```
##  $ grade  : num [1:2] 60 82
##  $ foreign: logi TRUE
```

We can accomplish indexing in `lists` using both `[]` and `$`.

```
str(j$grade)
##  num [1:2] 60 82
str(j[["grade"]])
##  num [1:2] 60 82
str(j[[2]])
##  num [1:2] 60 82
str(class(j[[2]]))
##  chr "numeric"
```

If we use `[]` instead `[[]]`, we would obtain another `list` instead of a `vector`.

```
str(j[2])
## List of 1
##  $ grade: num [1:2] 60 82
str(class(j[2]))
##  chr "list"
```

As in `matrices` and `vectors`, we can manipulate elements in `lists` using indexing techniques.

```
l <- list(name = "John",grade = 27)
str(l)
## List of 2
##  $ name : chr "John"
##  $ grade: num 27
l$info <- "foreign"
l[[4]] <- 95
str(l)
## List of 4
##  $ name : chr "John"
##  $ grade: num 27
##  $ info : chr "foreign"
##  $      : num 95
str(l[[3]])
##  chr "foreign"
l$grade <- NULL # delete z$b
str(l)
## List of 3
##  $ name: chr "John"
##  $ info: chr "foreign"
##  $     : num 95
str(l[[3]])
##  num 95
```

We can produce a `vector` from a `list` which contains all the atomic components using `unlist()`.

```
j <- list(name = c("Jane", "Mary"), grade = c(60, 82), foreign = T)
str(j)
## List of 3
##  $ name   : chr [1:2] "Jane" "Mary"
##  $ grade  : num [1:2] 60 82
##  $ foreign: logi TRUE
v <- unlist(j)
```

```
v
##   name1   name2  grade1  grade2 foreign
##  "Jane"  "Mary"    "60"    "82"  "TRUE"
names(v) <- NULL # delete names in vector
v
## [1] "Jane" "Mary" "60"   "82"   "TRUE"
unname(unlist(j)) # both unlist and delete names
## [1] "Jane" "Mary" "60"   "82"   "TRUE"
```

We can also create a recursive `list`.

```
homeworks <- list("1" = 45,"2" = 70)
exam <- list(exam = 65)
grade <- list(homeworks,exam)
str(grade)
## List of 2
##  $ :List of 2
##   ..$ 1: num 45
##   ..$ 2: num 70
##  $ :List of 1
##   ..$ exam: num 65
```

`data frame`

A `data frame` is a `matrix` like structure in which each column contains values of one variable, and each row contains one set of values from each column. We can create a `data frame` using `data.frame()`.

```
students <- c("Omar","Mark","Jane")
homework <- c(65,70,99)
exam <- c(45,90,70)
d <- data.frame(students,homework,exam,stringsAsFactors = FALSE)
str(d)
## 'data.frame':    3 obs. of  3 variables:
##  $ students: chr  "Omar" "Mark" "Jane"
##  $ homework: num  65 70 99
##  $ exam    : num  45 90 70
```

Filtering and indexing in `data frames` is a mixture of `list` and `matrix` objects.

```
str(d[[1]]) # obtain as vector
##  chr [1:3] "Omar" "Mark" "Jane"
str(d[1]) # obtain as data frame
## 'data.frame':    3 obs. of  1 variable:
##  $ students: chr  "Omar" "Mark" "Jane"
str(d$students)
##  chr [1:3] "Omar" "Mark" "Jane"
str(d[1:2,])
## 'data.frame':    2 obs. of  3 variables:
##  $ students: chr  "Omar" "Mark"
##  $ homework: num  65 70
##  $ exam    : num  45 90
str(d[1:2,2]) # reduced to vector
##  num [1:2] 65 70
str(d[1:2,2,drop = F]) # keep data frame structure
## 'data.frame':    2 obs. of  1 variable:
##  $ homework: num  65 70
```

When working with data frames, you will frequently come across `NA` values. The `na.rm = TRUE` argument and the `complete.cases()` function can make our life easier when working with `NA`.

```
d[2,3] <- NA
mean(d$height) # return NA because d[3,3] is NA
## [1] NA
mean(d$height,na.rm = TRUE) # compute mean of non-NA values
## [1] NA
complete.cases(d) # return a logical vector show which rows have no missing values
## [1]  TRUE FALSE  TRUE
str(d[complete.cases(d),])
## 'data.frame':    2 obs. of  3 variables:
##  $ students: chr  "Omar" "Jane"
##  $ homework: num  65 99
##  $ exam    : num  45 70
```

factor

`Factor` is the data structure that takes on a limited number of different values such variables are often referred to as categorical variables. It is best used to represent categorical variables when conducting data analysis. Both numeric and character variables can be made into a `factor`, but its levels will always be character values. A `factor` is stored as a `vector` of integer values with a corresponding set of character values. Use `str()` to further understand this. For more information, see here.

```
x <- c(15,20,25,20)
xf <- factor(x)
xf
## [1] 15 20 25 20
## Levels: 15 20 25
str(xf)
##  Factor w/ 3 levels "15","20","25": 1 2 3 2
unclass(xf)
## [1] 1 2 3 2
## attr(,"levels")
## [1] "15" "20" "25"
attr(xf,"levels")
## [1] "15" "20" "25"
length(xf)
## [1] 4
```

After creating a `factor`, it is not possible to add another value that is not shown at levels. You need to first add the new value to levels, then add the new value to the `factor`.

```
xf[3] <- 14
## Warning in `[<-.factor`(`*tmp*`, 3, value = 14): invalid factor level, NA
## generated
xf
## [1] 15   20   <NA> 20
## Levels: 15 20 25
xff <- factor(xf,levels = c(15,20,25,30))
xff[3] <- 14
## Warning in `[<-.factor`(`*tmp*`, 3, value = 14): invalid factor level, NA
## generated
xff
```

```
## [1] 15   20   <NA> 20
## Levels: 15 20 25 30
str(xff)
##  Factor w/ 4 levels "15","20","25",..: 1 2 NA 2
```

Control Statements

`if-else`

The syntax for `if` looks like this.

```
x <- 15
if(x > 0) {
  print(paste(x, "is a positive number"))
}
## [1] "15 is a positive number"
```

We can also add `else` after `if`.

```
x <- 7
if(x > 0) {
  print(paste(x, "is a positive number"))
} else {
  print(paste(x, "is a negative number or zero"))
}
## [1] "7 is a positive number"
```

If `if-else` includes one statement each, we can also use the compact form.

```
x <- 3
y <- if(x == 0) x else x+1
if(x == 0) y <- x else y <- x+1
print(paste("x = ", x, "and y = ", y))
## [1] "x =  3 and y =  4"
```

We can also use `else if` after `if`. After `else if`, we can also use `else` or `else if` again but these are not necessary.

```
x <- 3
if (x > 0) {
  print(paste(x, "is a positive number"))
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} # option 1
## [1] "3 is a positive number"
x <- 0
if (x > 0) {
  print(paste(x, "is a positive number"))
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} else if (x == 0) {
  print(paste(x, "is zero"))
} # option 2
## [1] "0 is zero"
if (x > 0) {
  print(paste(x, "is a positive number"))
```

```
} else if (x < 0) {
  print(paste(x, "is a negative number"))
} else {
  print(paste(x, "is zero"))
} # option 3
## [1] "0 is zero"
```

Loops

The most frequently used loop, `for`, looks like this.

```
for (i in 2015:2020) {
  print(paste("The year is", i))
}
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
## [1] "The year is 2018"
## [1] "The year is 2019"
## [1] "The year is 2020"
```

We have one statement, we can also use the compact form.

```
for (i in 2015:2020) print(paste("The year is", i))
## [1] "The year is 2015"
## [1] "The year is 2016"
## [1] "The year is 2017"
## [1] "The year is 2018"
## [1] "The year is 2019"
## [1] "The year is 2020"
```

Another loop style is `while`.

```
i <- 2015
while (i < 2021) {
  print(i)
  i <- i+1
}
## [1] 2015
## [1] 2016
## [1] 2017
## [1] 2018
## [1] 2019
## [1] 2020
```

A typical looping sequence can be altered using the `break` or the `next` statement. A `break` statement is used inside a loop to stop the iterations and flow the control outside of the loop. A `next` statement is useful when we want to skip the current iteration of a loop without terminating it.

```
for (i in 1:10) {
  if (!i %% 2){
    next
  }
  print(i)
}
## [1] 1
```

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

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

The last `R` loop is the `repeat` loop.

```
x <- 5
repeat {
if (x < 1){
  print(paste(x, "is not a positive number"))
break
}
  print(paste(x, "is a positive number"))
  x <- x-1
}
## [1] "5 is a positive number"
## [1] "4 is a positive number"
## [1] "3 is a positive number"
## [1] "2 is a positive number"
## [1] "1 is a positive number"
## [1] "0 is not a positive number"
```

Functions in `R`

Up to this point, we use `R` built-in functions to do some basic operations on data structures. Instead of using `R` built in functions, we can also create our custom functions. In general, a function needs a name, arguments, and a return value. A function will return the value of the last statement executed unless a return statement is explicitly called.

```
is.prime <- function(x) { # function for checking whether a number is prime
  y <- T
  if (x==2) {
    y <- T
  } else if (x<2) {
    warning("input should be equal to or greater than 2 \n")
    y <- "input should be equal to or greater than 2"
  } else {
    for(i in 2:ceiling(x / 2)) {
      if(x %% i == 0) {
        y <- F
        break
      }
      }
```

```
    }
return(y)
}
for (i in 1:20) {
  if (is.prime(i)==T) print(paste(i, "is a prime number"))
}
## Warning in is.prime(i): input should be equal to or greater than 2
## [1] "2 is a prime number"
## [1] "3 is a prime number"
## [1] "5 is a prime number"
## [1] "7 is a prime number"
## [1] "11 is a prime number"
## [1] "13 is a prime number"
## [1] "17 is a prime number"
## [1] "19 is a prime number"
```

Let's show another function example with multiple parameters and also learn how to deal with error with `tryCatch()`.

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  # stopifnot(): If the expression is not true, break and produce an error message
  stopifnot(length(x) == length(w))
  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
    }
  sum(w * x) / sum(x)
  }
tryCatch( {
  wt_mean(1:6, 6:1, na.rm = A) # function to be executed
}, error=function(cond) { # error message to catch
  message("There is an error") # custom error message to produce
  message(cond) # actual error message
}, warning=function(cond) { # warning message to catch
    message("There is a warning") # warning message to produce
  message(cond) # actual warning message
}) # error 1
## There is an error
tryCatch( {
  wt_mean(1:5, 6:1, na.rm = T)
}, error=function(cond) {
  message("There is an error")
  message(cond)
}, warning=function(cond) {
    message("There is a warning")
  message(cond)
}) # error 2
## There is an error
tryCatch( {
  wt_mean(1:6, 6:1)
}, error=function(cond) {
  message("There is an error")
```

```r
  message(cond)
}, warning=function(cond) {
    message("There is a warning")
  message(cond)
}) # success
## [1] 2.666667
```

Functions can also take arbitrary number of arguments using a special argument: `...`.

```r
commas <- function(...) {
  paste(..., collapse = ", ")
}
commas(letters[1:10])
## [1] "a, b, c, d, e, f, g, h, i, j"
```

`apply()` family in `R`

`apply()` returns a `vector`, `array` or `list` of values obtained by applying a function to margins of an `data frame`, a `matrix`, or an `array` and is primarily used to avoid explicit loops.

```r
students <- c("Omar","Mark","Jane")
homework <- c(65,70,99)
exam <- c(45,90,70)
d <- data.frame(homework,exam,stringsAsFactors = FALSE, row.names = students)
str(d)
## 'data.frame':    3 obs. of  2 variables:
##  $ homework: num  65 70 99
##  $ exam    : num  45 90 70
apply(d, 2, mean) # column mean
## homework     exam
## 78.00000 68.33333
apply(d, 1, mean) # row mean
## Omar Mark Jane
## 55.0 80.0 84.5
f <- function(x) sum(x)/length(x)
apply(d, 2, f) # column mean
## homework     exam
## 78.00000 68.33333
apply(d, 1, f) # row mean
## Omar Mark Jane
## 55.0 80.0 84.5
```

`lapply()` is used to apply a function to all the elements of a `list`, a `data frame`, or a `vector`. It produces a `list` as output.

```r
j <- list(name = c("Joe", "Mary"), sex = c("Male", "Female"))
str(j)
## List of 2
##  $ name: chr [1:2] "Joe" "Mary"
##  $ sex : chr [1:2] "Male" "Female"
j_upper <-lapply(j, toupper)
str(j_upper)
## List of 2
##  $ name: chr [1:2] "JOE" "MARY"
##  $ sex : chr [1:2] "MALE" "FEMALE"
```

sapply() works as lapply(), but produces a vector or a matrix as output instead of a list.

```
j <- list(name = c("Joe", "Mary"), sex = c("Male", "Female"))
str(j)
## List of 2
##  $ name: chr [1:2] "Joe" "Mary"
##  $ sex : chr [1:2] "Male" "Female"
j_upper <-sapply(j, toupper)
str(j_upper)
##  chr [1:2, 1:2] "JOE" "MARY" "MALE" "FEMALE"
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "name" "sex"
```

The operation performed by tapply(x, f, g) is to (temporarily) split x into groups, each group corresponding to a level of the factor (or a combination of levels of the factors in the case of multiple factors), and then apply g() to the resulting subvectors of x.

```
grade <- c(90,75,60,45,36,24)
sex <- c("F","F","M","M","F","M")
str(tapply(grade,sex,mean))
##  num [1:2(1d)] 67 43
##  - attr(*, "dimnames")=List of 1
##   ..$ : chr [1:2] "F" "M"
```

tidyverse

After introducing base R, we move to the tidyverse environment. The tidyverse is an opinionated collection of R packages designed for conducting data science. All packages under the tidyverse umbrella share an underlying design philosophy, grammar, and data structures. If you have not installed the tidyverse packages yet, it is a good time to install it now.

```
if (!"tidyverse" %in% rownames(installed.packages())) install.packages("tidyverse")
require("tidyverse")
## Loading required package: tidyverse
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

dplyr

First, we will start with the dplyr package. dplyr is the grammar of data manipulation in tidyverse providing a consistent set of verbs that help you solve the most common data manipulation challenges. We will use the COVID-19 data set of CSSEGISandData to show dplyr verbs. To keep it simple, we will only use the last two days of the data set for now. To load the data set, we use the read_csv() function from the tidyverse environment. read_csv() read the csv file and convert to a tibble. In tidyverse environment, we will mostly use the tibble data structure. It is the modern version of data frame that improves data frame in a number of ways: it never changes an input's type; it never adjusts the names of variables; it evaluates its arguments lazily and sequentially; it never uses row.names(); it only recycles vectors of length 1.

```r
Raw_Data <- read_csv("https://tinyurl.com/tsqkf7y")
Data <- Raw_Data[,c(1,2,3,4,ncol(Raw_Data)-1,ncol(Raw_Data))]
head(Data, 3)
## # A tibble: 3 x 6
##   `Province/State` `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>            <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 <NA>             Afghanistan       33.9 67.7      47306     47516
## 2 <NA>             Albania           41.2 20.2      42988     43683
## 3 <NA>             Algeria           28.0  1.66     88252     88825
```

We start with the first fundamental verb of `dplyr`: `filter()` which picks cases based on their values. We first filter out Germany, then we filter out both Germany and Austria, together.

```r
filter(Data, `Country/Region`== "Germany")
## # A tibble: 1 x 6
##   `Province/State` `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>            <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 <NA>             Germany           51.2  10.5   1194550   1200006
filter(Data, `Country/Region` %in% c("Germany","Austria"))
## # A tibble: 2 x 6
##   `Province/State` `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>            <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 <NA>             Austria           47.5  14.6    303430    305693
## 2 <NA>             Germany           51.2  10.5   1194550   1200006
```

The second fundamental verb of `dplyr` is `arrange()`, which changes the ordering of the rows. First we order data according to latitude. Second we order according to the 6th column. We can also order data according to more than one column.

```r
head(arrange(Data, Lat), 2)
## # A tibble: 2 x 6
##   `Province/State`             `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>                        <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 Falkland Islands (Malvinas) United Kingdom   -51.8 -59.5        17        17
## 2 Tasmania                     Australia        -42.9 147.        230       230
head(arrange(Data, 6), 2)
## # A tibble: 2 x 6
##   `Province/State` `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>            <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 <NA>             Afghanistan       33.9  67.7     47306     47516
## 2 <NA>             Albania           41.2  20.2     42988     43683
head(arrange(Data, Lat, Long), 2) # this doesn't make any changes in our case
## # A tibble: 2 x 6
##   `Province/State`             `Country/Region`   Lat  Long `12/6/20` `12/7/20`
##   <chr>                        <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 Falkland Islands (Malvinas) United Kingdom   -51.8 -59.5        17        17
## 2 Tasmania                     Australia        -42.9 147.        230       230
```

The third fundamental verb of `dplyr` is `select()`, which picks variables based on their names. We can select columns one by one or use `:` to select multiple columns.

```r
head(select(Data, Lat, Long),2)
## # A tibble: 2 x 2
##     Lat  Long
##   <dbl> <dbl>
```

```
## 1  33.9  67.7
## 2  41.2  20.2
head(select(Data, `Province/State`: Long),2)
## # A tibble: 2 x 4
##   `Province/State` `Country/Region`   Lat  Long
##   <chr>            <chr>            <dbl> <dbl>
## 1 <NA>             Afghanistan       33.9  67.7
## 2 <NA>             Albania           41.2  20.2
```

We can select columns also by its position or make it the first column.

```
head(select(Data, 6), 2)
## # A tibble: 2 x 1
##   `12/7/20`
##       <dbl>
## 1     47516
## 2     43683
head(select(Data, 6, everything()), 2)
## # A tibble: 2 x 6
##   `12/7/20` `Province/State` `Country/Region`   Lat  Long `12/6/20`
##       <dbl> <chr>            <chr>            <dbl> <dbl>     <dbl>
## 1     47516 <NA>             Afghanistan       33.9  67.7     47306
## 2     43683 <NA>             Albania           41.2  20.2     42988
```

Finally, we can delete columns. We don't need the columns `Lat`, `Long`, so we drop them.

```
Data <- select(Data, -(Lat:Long))
head(Data, 2)
## # A tibble: 2 x 4
##   `Province/State` `Country/Region` `12/6/20` `12/7/20`
##   <chr>            <chr>                <dbl>     <dbl>
## 1 <NA>             Afghanistan          47306     47516
## 2 <NA>             Albania              42988     43683
```

To rename variables, we can use a variant of `select()`: `rename()`.

```
Data <- rename(Data, Last_Day = names(select(Data, last_col())),
               Previous_Day = names(select(Data, last_col()-1)),
               Country = `Country/Region`, SubRegion = `Province/State`)
head(Data, 2)
## # A tibble: 2 x 4
##   SubRegion Country     Previous_Day Last_Day
##   <chr>     <chr>              <dbl>    <dbl>
## 1 <NA>      Afghanistan        47306    47516
## 2 <NA>      Albania            42988    43683
```

The fourth fundamental verb of `dplyr` is `mutate()`, which adds new variables that are functions of existing variables. We will create the `New_Cases` column as the difference of `Last_Day` and `Previous_Day`.

```
Data_New <- mutate(Data, New_Cases = Last_Day-Previous_Day)
head(arrange(Data_New, desc(New_Cases)), 2)
## # A tibble: 2 x 5
##   SubRegion Country Previous_Day Last_Day New_Cases
##   <chr>     <chr>          <dbl>    <dbl>     <dbl>
## 1 <NA>      US          14757000 14949299    192299
## 2 <NA>      Turkey        828295   860432     32137
```

To remove all variables except the newly created variables, use `transmute()`.

```
head(transmute(Data, New_Cases=Last_Day-Previous_Day), 2)
## # A tibble: 2 x 1
##   New_Cases
##       <dbl>
## 1       210
## 2       695
```

Useful creation functions used with `mutate()`: `arithmetic operators`; `modular arithmetic`; `logical comparisions`; `log()`; `log2()`; `log10()`; `lead()`; `lag()`; `cumsum()`; `cumprod()`; `cummin()`; `cummax()`; `cummean()`; `min_rank()`; `row_number()`; `dense_rank()`; `percent_rank()`; `cume_dist()`.

The last fundamental function of `dplyr` is `summarize()`, which reduces multiple values down to a single summary. Useful summary functions: `mean()`; `median()`; `sd()`; `IQR()`; `mad()`; `min()`; `quantile()`; `max()`; `first()`; `nth()`; `last()`; `n()`; `sum(x > 10)`.

```
summarize(Data_New, Total_Cases = sum(New_Cases, na.rm = TRUE))
## # A tibble: 1 x 1
##   Total_Cases
##         <dbl>
## 1      517473
```

All fundamental verbs mentioned above can be combined naturally with `group_by()` which allows you to perform any operation by group. Let's aggregate subregions by using the `group_by()` function.

```
by_country <- group_by(Data_New, Country)
Agg_Data <- summarize(by_country, New_Cases = sum(New_Cases, na.rm = TRUE),
                      Last_Day = sum(Last_Day, na.rm = TRUE),
                      Previous_Day = sum(Previous_Day, na.rm = TRUE), Count = n())
head(Agg_Data, 2)
## # A tibble: 2 x 5
##   Country     New_Cases Last_Day Previous_Day Count
##   <chr>           <dbl>    <dbl>        <dbl> <int>
## 1 Afghanistan       210    47516        47306     1
## 2 Albania           695    43683        42988     1
```

tidyr

The goal of `tidyr` is to help you creating tidy data in which every column is variable, every row is an observation, and every cell is a single value. One of the fundamental functions of `tidyr` is `gather()`, which gathers columns into a new pair of variables. In the following code, we also use the pipe operator (`%>%`) from the `magrittr` package. `%>%` takes the output of one statement and makes it the input of the next statement.

```
Tidy_Data <- Agg_Data %>%
  gather(Last_Day, Previous_Day, key = "Date",
         value = "Cumulative_Cases", na.rm = TRUE)
head(Tidy_Data, 3)
## # A tibble: 3 x 5
##   Country     New_Cases Count Date     Cumulative_Cases
##   <chr>           <dbl> <int> <chr>               <dbl>
## 1 Afghanistan       210     1 Last_Day            47516
## 2 Albania           695     1 Last_Day            43683
## 3 Algeria           573     1 Last_Day            88825
```

Another important function is `spread()` which is the opposite of gathering. `gather()` makes wide tables narrower and longer, whereas `spread()` makes long tables shorter and wider.

```r
Spread_Data <- Tidy_Data %>%
  spread(key = "Date", value="Cumulative_Cases")
head(Spread_Data, 3)
## # A tibble: 3 x 5
##   Country     New_Cases Count Last_Day Previous_Day
##   <chr>           <dbl> <int>    <dbl>        <dbl>
## 1 Afghanistan       210     1    47516        47306
## 2 Albania           695     1    43683        42988
## 3 Algeria           573     1    88825        88252
```

`unite()` is the function that combines multiple columns into a single column. In `Spread_Data`, let's combine `Last_Day` and `Previous_Day` into `Case_Comparision` column

```r
Unite_Date <- Spread_Data %>%
  unite(Case_Comparision, Last_Day, Previous_Day,  sep = "/")
head(Unite_Date, 3)
## # A tibble: 3 x 4
##   Country     New_Cases Count Case_Comparision
##   <chr>           <dbl> <int> <chr>
## 1 Afghanistan       210     1 47516/47306
## 2 Albania           695     1 43683/42988
## 3 Algeria           573     1 88825/88252
```

To split columns, we can use `separate()`.

```r
Unite_Date %>%
  separate(Case_Comparision, into = c("Last_Day", "Previous_Day"), sep = "/") %>%
  head(3)
## # A tibble: 3 x 5
##   Country     New_Cases Count Last_Day Previous_Day
##   <chr>           <dbl> <int> <chr>    <chr>
## 1 Afghanistan       210     1 47516    47306
## 2 Albania           695     1 43683    42988
## 3 Algeria           573     1 88825    88252
```

Other important functions of `tidyr`, especially to deal with the missing values, are `complete()` and `fill()`. `complete()` turns implicit missing values into explicit missing values, and `fill()` replaces missing values in selected columns with the next or previous entry. Let's give an example about these functions.

```r
df <- tibble(
  group = c(1:2,1),
  item_name = c("a","b","b"), value1 = 1:3, value2 = 4:6)
head(df,3)
## # A tibble: 3 x 4
##   group item_name value1 value2
##   <dbl> <chr>      <int>  <int>
## 1     1 a              1      4
## 2     2 b              2      5
## 3     1 b              3      6
dfc <- df %>% complete(group, nesting(item_name))
head(dfc,4)
## # A tibble: 4 x 4
##   group item_name value1 value2
##   <dbl> <chr>      <int>  <int>
## 1     1 a              1      4
## 2     1 b              3      6
```

```
## 3     2 a             NA     NA
## 4     2 b              2      5
dfc %>% fill(value1, value2) %>% head(4)
## # A tibble: 4 x 4
##   group item_name value1 value2
##   <dbl> <chr>      <int>  <int>
## 1     1 a              1      4
## 2     1 b              3      6
## 3     2 a              3      6
## 4     2 b              2      5
```

Relational Data with `dplyr`

After learning the fundamentals of `dplyr` and `tidyr`, we can now focus on relational data. When conducting data analysis, you will work with many data sets and you need to merge them in many cases. Multiple tables of data are called relational data and relations are always defined between a pair of tables. Let's use the population data for 2019 provided by World Bank (WB).

```
require("WDI") # world bank data base
require("countrycode") # manipulate country codes
pop <- WDIsearch("Population, total")
WB_Data <- WDI(indicator = pop[1], start = 2019, end = 2019)
# for easier merging obtain iso3 country codes for WB_Data
WB_Data$iso3c <- countrycode(WB_Data[,1], origin ='iso2c', destination = 'iso3c')
head(WB_Data, 3)
##   iso2c                        country indicator year iso3c
## 1    1A                     Arab World 427870270 2019  <NA>
## 2    S3         Caribbean small states   7401381 2019  <NA>
## 3    B8 Central Europe and the Baltics 102378579 2019  <NA>
# for easier merging obtain iso3 country codes for Tidy_Data
# we use pull() to convert tibble to vector as countrycode() accepts that format
Tidy_Data$iso3c <- countrycode(pull(Tidy_Data, Country), origin = 'country.name', destination = 'iso3c')
head(Tidy_Data, 3)
## # A tibble: 3 x 6
##   Country     New_Cases Count Date     Cumulative_Cases iso3c
##   <chr>           <dbl> <int> <chr>               <dbl> <chr>
## 1 Afghanistan       210     1 Last_Day            47516 AFG
## 2 Albania           695     1 Last_Day            43683 ALB
## 3 Algeria           573     1 Last_Day            88825 DZA
```

We want to combine estimated population data for countries with our existing data set to calculate case numbers per capita. We will merge population data with our tidy data using `left_join()`.

```
Merged_Data <- Tidy_Data %>%
  left_join(WB_Data, by = "iso3c") %>%
  rename(Population = indicator) %>%
  mutate(New_Cases_per_Capita = New_Cases/Population*1000000) %>%
  mutate(Cumulative_Cases_per_Capita = Cumulative_Cases/Population*1000000) %>%
  select(iso3c, Date, New_Cases, New_Cases_per_Capita, Cumulative_Cases, Cumulative_Cases_per_Capita)
head(Merged_Data, 4)
## # A tibble: 4 x 6
##   iso3c Date   New_Cases New_Cases_per_Cap~ Cumulative_Cases Cumulative_Cases_p~
##   <chr> <chr>      <dbl>              <dbl>            <dbl>               <dbl>
## 1 AFG   Last_~       210               5.52            47516               1249.
## 2 ALB   Last_~       695             244.              43683              15305.
```

```
## 3 DZA   Last_~       573              13.3              88825              2063.
## 4 AND   Last_~        34             441.                7084             91831.
```

As some country's names are different between our data set and IMF data set, there are unmatched countries. To keep only the matched keys, we can use `inner_join()`.

```
Merged_Data <- Tidy_Data %>%
  inner_join(WB_Data, by = "iso3c") %>%
  select(iso3c, Country, Date, New_Cases, Cumulative_Cases, indicator) %>%
  rename(Population = indicator)
head(Merged_Data, 4)
## # A tibble: 4 x 6
##   iso3c Country     Date     New_Cases Cumulative_Cases Population
##   <chr> <chr>       <chr>        <dbl>            <dbl>      <dbl>
## 1 AFG   Afghanistan Last_Day       210            47516   38041754
## 2 ALB   Albania     Last_Day       695            43683    2854191
## 3 DZA   Algeria     Last_Day       573            88825   43053054
## 4 AND   Andorra     Last_Day        34             7084      77142
```

We can also use `full_join()` to keep all observations in both data sets.

```
Merged_Data <- Tidy_Data %>%
  full_join(WB_Data, by= "iso3c") %>%
  select(iso3c, Country, Date, New_Cases, Cumulative_Cases, indicator) %>%
  rename(Population = indicator)
tail(Merged_Data, 4)
## # A tibble: 4 x 6
##   iso3c Country Date  New_Cases Cumulative_Cases Population
##   <chr> <chr>   <chr>     <dbl>            <dbl>      <dbl>
## 1 TKM   <NA>    <NA>         NA               NA    5942089
## 2 TCA   <NA>    <NA>         NA               NA      38191
## 3 TUV   <NA>    <NA>         NA               NA      11646
## 4 VIR   <NA>    <NA>         NA               NA     106631
```

If two data sets have different names for key column, we need to use `left_join(X, Y, c("Key_X"="Key_Y)`. We can also use `semi_join(x, y)` and `anti_join(x, y)` for filtering. `semi_join(x, y)` keeps all observations in x that have a match in y and `anti_join(x, y)` drops all observations in x that have a match in y.

```
dim(Tidy_Data)
## [1] 384   6
dim(anti_join(Tidy_Data, WB_Data))
## [1] 6 6
dim(semi_join(Tidy_Data, WB_Data))
## [1] 378   6
```

Finally, there are three set operations: `intersect(x, y)` that returns only observation both in x and y; `union(x, y)` that returns unique observations in x and y; `setdiff(x, y)` that returns observations in x, but not in y. Let give one example about set operations:

```
mtcars$model <- rownames(mtcars)
first <- mtcars[1:20, ]
second <- mtcars[10:32, ]

dim(intersect(first, second))
## [1] 11 12
```

```
dim(union(first, second))
## [1] 32 12
dim(setdiff(first, second))
## [1]  9 12
```

lubridate

It is usually hard to work with date and date time data in `R`. `lubridate` makes it easier to create and manipulate date and date time data in `R`. We can create date and date-time from strings such as `ymd("2020-12-31")`, `mdy("December 31st, 2020")`, `dmy("31-Dec-2020")`, `ymd(20201231)`, `ymd_hms("2020-12-31 23:59:59")`, `mdy_hm("12/31/2020 08:01")`. We can also use `make_datetime(year = 2020, month = 12, day = 31, hour = 23, min = 59, sec = 59, tz = "UTC")` and `make_date(year = 2020, month = 12, day = 31)` to create `date` and `date-time`. Instead of strings let's create our date data from rows of our data set.

```
head(Raw_Data[,c(1,2,3,4,5,6)], 3)
## # A tibble: 3 x 6
##   `Province/State` `Country/Region`   Lat  Long `1/22/20` `1/23/20`
##   <chr>            <chr>            <dbl> <dbl>     <dbl>     <dbl>
## 1 <NA>             Afghanistan       33.9 67.7          0         0
## 2 <NA>             Albania           41.2 20.2          0         0
## 3 <NA>             Algeria           28.0  1.66         0         0
Large_Tidy_Data <- Raw_Data %>%
  gather("1/22/20":names(Raw_Data[,ncol(Raw_Data)]),
         key = "Date", value = "Cumulative_Cases", na.rm = TRUE) %>%
  mutate(Date=as.Date(Date, format = "%m/%d/%y")) %>%
  rename(Country = `Country/Region`, SubRegion = `Province/State`) %>%
  select(Date, SubRegion, Country, Cumulative_Cases) %>%
  group_by(Country, Date) %>%
  summarize(New_Cases = sum(Cumulative_Cases, na.rm = TRUE), Count = n()) %>%
  arrange(Date)
head(Large_Tidy_Data, 3)
## # A tibble: 3 x 4
## # Groups:   Country [3]
##   Country     Date       New_Cases Count
##   <chr>       <date>         <dbl> <int>
## 1 Afghanistan 2020-01-22         0     1
## 2 Albania     2020-01-22         0     1
## 3 Algeria     2020-01-22         0     1
```

To get individual parts of the date, we can use `year(datetime)`, `month(datetime)`, `mday(datetime)`, `yday(datetime)`, `wday(datetime)`. We can also update each individual part of the datetime such as `year(datetime) <- 2021`, `month(datetime) <- 01`, `hour(datetime) <- hour(datetime) + 1`, `update(datetime, year = 2021, month = 1, mday = 1, hour = 1)`.

```
str(lubridate:::year(Large_Tidy_Data$Date)) # Extract year
##  num [1:61632] 2020 2020 2020 2020 2020 2020 2020 2020 2020 2020 ...
str(lubridate:::month(Large_Tidy_Data$Date)) # Extract month
##  num [1:61632] 1 1 1 1 1 1 1 1 1 1 ...
str(lubridate:::day(Large_Tidy_Data$Date)) # Extract day
##  int [1:61632] 22 22 22 22 22 22 22 22 22 22 ...
```

For time spans, `lubridate` offers `durations`, which represent an exact number of seconds and `periods`, which represent human units like weeks and months. Important `duration` constructors are `as.duration(today() - ymd(19841119))`, `dseconds(50)`, `dminutes(50)`, `dhours(c(36, 72))`, `ddays(3:9)`, `dweeks(2)`, `dyears(2)`.

You can add and subtract `durations` to and from days: `tomorrow <- today() + ddays(3)`. Important `periods` constructors are `seconds(50)`, `minutes(50)`, `hours(c(36, 72))`, `days(5)`, `months(5:10)`, `weeks(5)`, `years(5)`. You can add and subtract `periods` to and from days: `ymd("2020-12-31") + years(2)`.

```
lubridate:::as.duration(lubridate:::today()-Large_Tidy_Data$Date[1]) # how many seconds
## [1] "27734400s (~45.86 weeks)"
# has passed since the first day in our data set
lubridate:::as.period(lubridate:::today()-Large_Tidy_Data$Date[1]) # how many days
## [1] "321d 0H 0M 0S"
# has passed since the first day in our data set
```

purrr

Before moving to `purrr` package, let's show how to perform `for` loops on `tibble` objects with two examples.

```
set.seed(1)
df <- tibble(a = rnorm(10), b = rnorm(10), c = rnorm(10), d = rnorm(10))
str(df)
## tibble [10 x 4] (S3: tbl_df/tbl/data.frame)
##  $ a: num [1:10] -0.626 0.184 -0.836 1.595 0.33 ...
##  $ b: num [1:10] 1.512 0.39 -0.621 -2.215 1.125 ...
##  $ c: num [1:10] 0.919 0.7821 0.0746 -1.9894 0.6198 ...
##  $ d: num [1:10] 1.3587 -0.1028 0.3877 -0.0538 -1.3771 ...
output <- vector("double", ncol(df))
for (i in seq_along(df)) {
  output[[i]] <- median(df[[i]])
}
str(output)
##  num [1:4] 0.25658 0.49187 0.00922 -0.05656
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
        out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
## [1]  0.256575548  0.491872279  0.009218122 -0.056559219
col_summary(df, mean)
## [1]  0.1322028  0.2488450 -0.1336732  0.1207302
```

Instead of explicit `for` loops, the `purrr` package provides a series of functions similar to `sapply()` and `lapply()`: `map()`, which makes a `list`; `map_lgl()`, which makes a logical `vector`; `map_int()`, which makes an integer `vector`; `map_dbl()`, which makes a double `vector`; and `map_chr()`, which makes a character vector. These functions takes a `vector` as input, applies a function to each piece, and then returns a new `vector` that's the same length as the input.

```
map_dbl(df, mean)
##          a          b          c          d
##  0.1322028  0.2488450 -0.1336732  0.1207302
```

For dealing with errors, we can use the `safely()` function which provides both the result and the error element. Other similar functions are `possibly()`, which gives a default value to return when there is an error, and `quietly()`, which captures printed output, messages, and warnings.

```r
x <- list(0, 10, "a")
y <- x %>% map(safely(log))
str(y)
## List of 3
##  $ :List of 2
##   ..$ result: num -Inf
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 2.3
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: NULL
##   ..$ error :List of 2
##   .. ..$ message: chr "non-numeric argument to mathematical function"
##   .. ..$ call   : language .Primitive("log")(x, base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
y <- x %>% map(possibly(log, NA))
str(y)
## List of 3
##  $ : num -Inf
##  $ : num 2.3
##  $ : logi NA
x <- list(1, -1)
x %>% map(quietly(log)) %>% str()
## List of 2
##  $ :List of 4
##   ..$ result  : num 0
##   ..$ output  : chr ""
##   ..$ warnings: chr(0)
##   ..$ messages: chr(0)
##  $ :List of 4
##   ..$ result  : num NaN
##   ..$ output  : chr ""
##   ..$ warnings: chr "NaNs produced"
##   ..$ messages: chr(0)
```

For iterating over two arguments we can use `map2()` and for more than two arguments, we can use the `pmap()` function.

```r
set.seed(1)
mu <- list(5, 10, -3)
sigma <- list(1, 5, 10)
map2(mu, sigma, rnorm, n = 5) %>% str()
## List of 3
##  $ : num [1:5] 4.37 5.18 4.16 6.6 5.33
##  $ : num [1:5] 5.9 12.44 13.69 12.88 8.47
##  $ : num [1:5] 12.118 0.898 -9.212 -25.147 8.249
n <- list(1, 3, 5)
args1 <- list(mean = mu, sd = sigma, n = n)
str(args1)
## List of 3
##  $ mean:List of 3
##   ..$ : num 5
##   ..$ : num 10
```

```
##   ..$ : num -3
##  $ sd  :List of 3
##   ..$ : num 1
##   ..$ : num 5
##   ..$ : num 10
##  $ n   :List of 3
##   ..$ : num 1
##   ..$ : num 3
##   ..$ : num 5
args1 %>% pmap(rnorm) %>% str()
## List of 3
##  $ : num 4.96
##  $ : num [1:3] 9.92 14.72 14.11
##  $ : num [1:5] 2.94 6.19 4.82 -2.25 -22.89
```

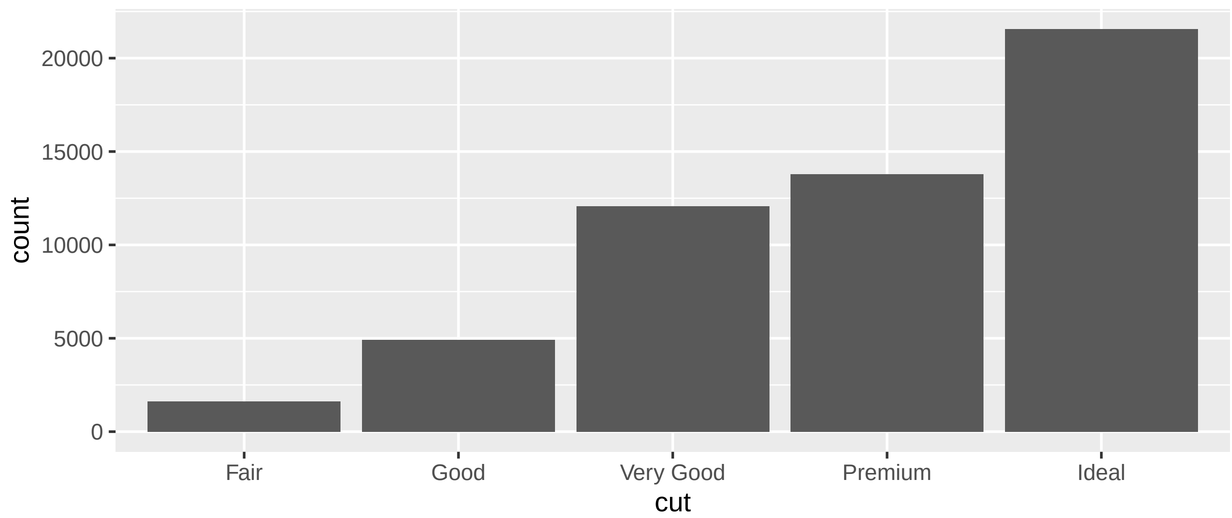
For iterating multiple functions, we can use `invoke_map()`.

```
f <- c("runif", "rnorm")
param <- list(list(min = -1, max = 1), list(sd = 5))
invoke_map(f, param, n=5) %>% str()
## List of 2
##  $ : num [1:5] 0.4646 0.3855 -0.0448 0.7224 -0.1238
##  $ : num [1:5] -3.455 -6.423 0.234 -1.179 -2.714
```

ggplot2

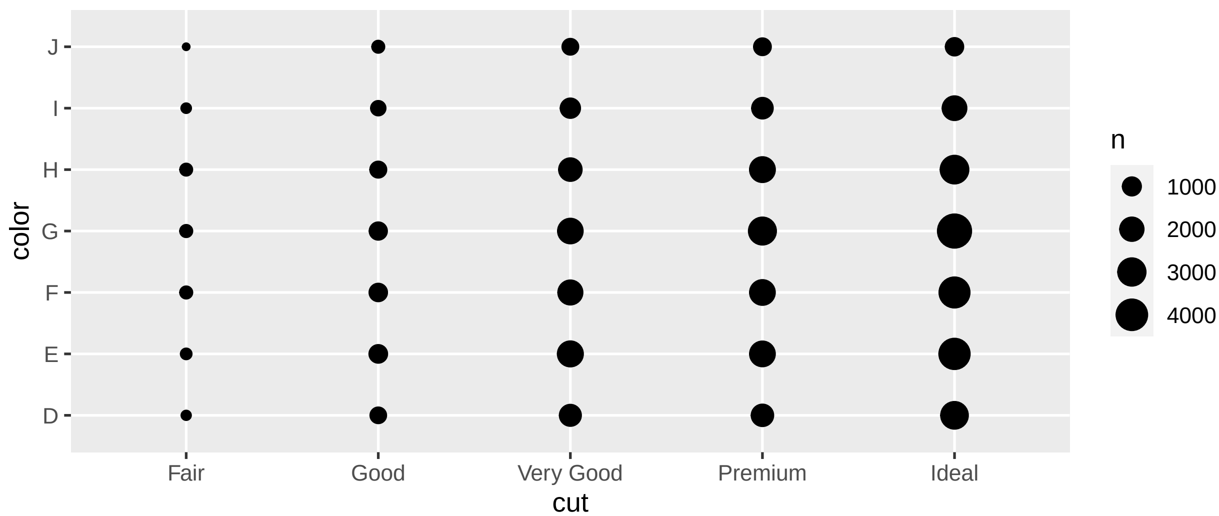
The distribution of a variable will depend on whether the variable is categorical or continuous. We will first use a categorical variable. Let's create a data set including diamonds with a size of less than three carats and create our first bar chart with `geom_bar()`.

```
smaller <- diamonds %>%
  filter(carat < 3)
head(diamonds, 3)
## # A tibble: 3 x 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good    E     VS1      56.9    65   327  4.05  4.07  2.31
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

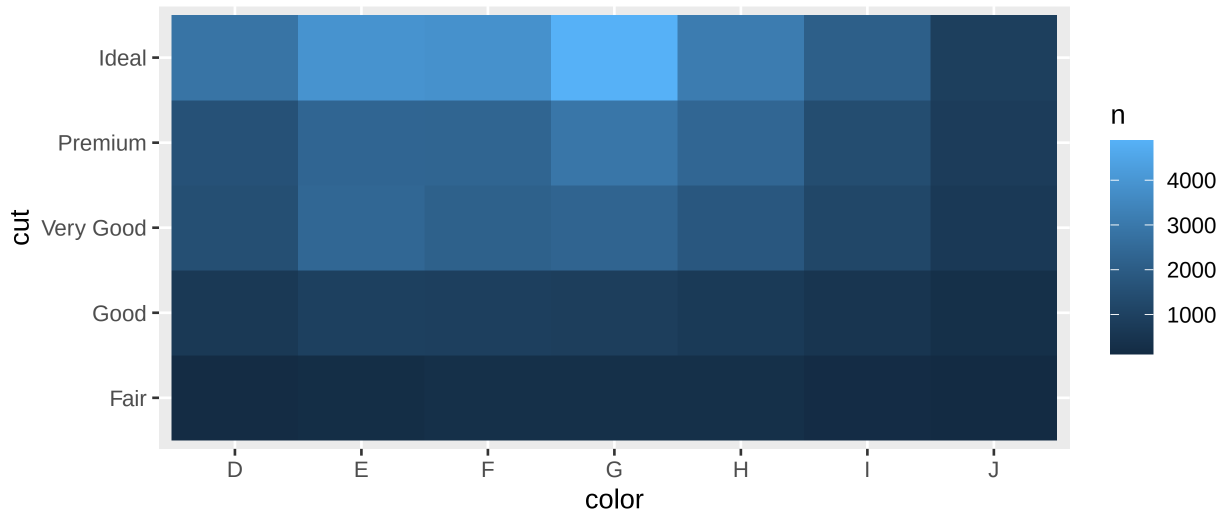


We can also visualize the covariation between categorical variables by counting the number of observations for each combination by using `geom_count()` or `geom_tile()` and `count()`.

```
ggplot(data = diamonds) +
  geom_count(mapping = aes(x = cut, y = color))
```

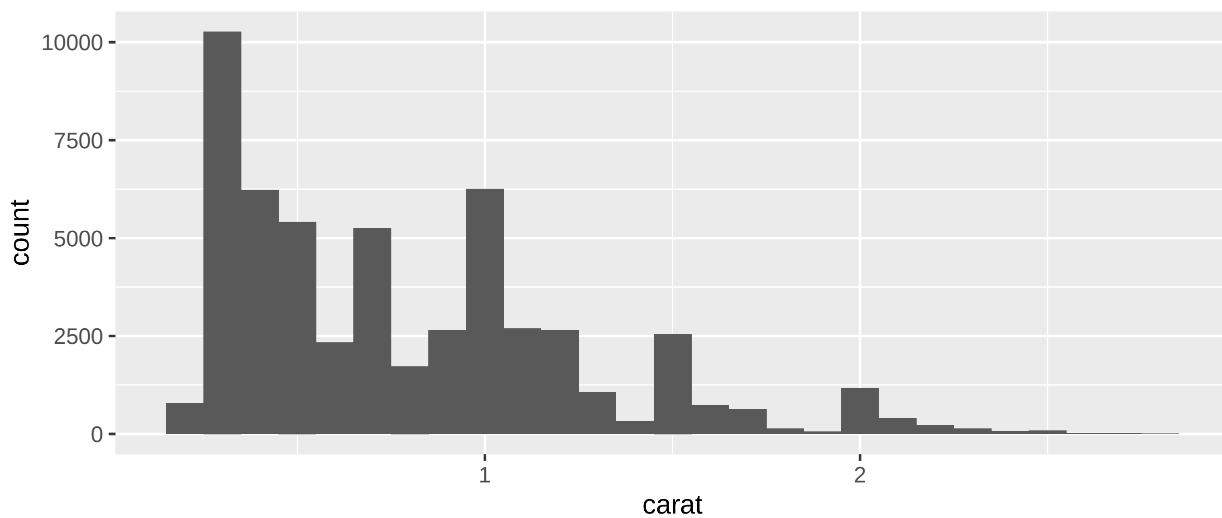



```
diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
    geom_tile(mapping = aes(fill = n))
```



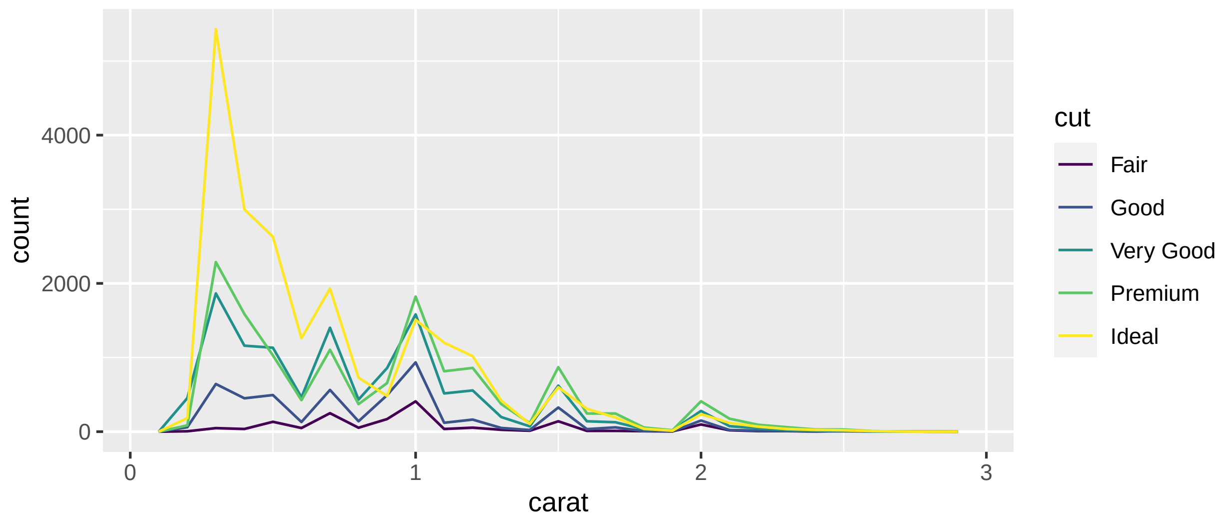
Next, we will use a continuous variable and draw a histogram with `geom_histogram()`.

```
ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
```



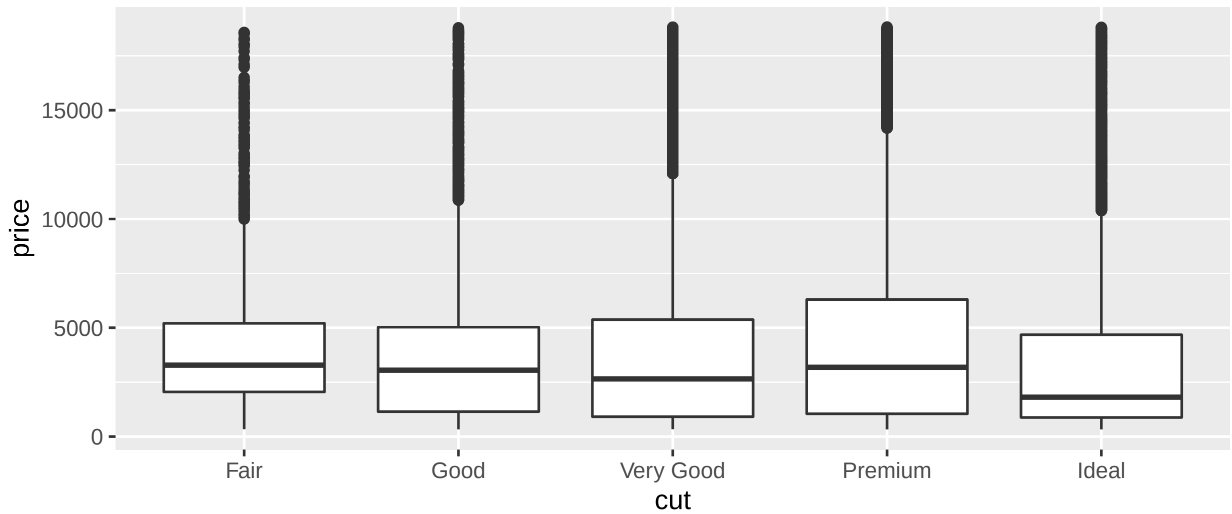
We can also use `geom_freqpoly()` which displays the same information with lines.

```
ggplot(data = smaller, mapping = aes(x = carat, color = cut)) +
  geom_freqpoly(binwidth = 0.1)
```

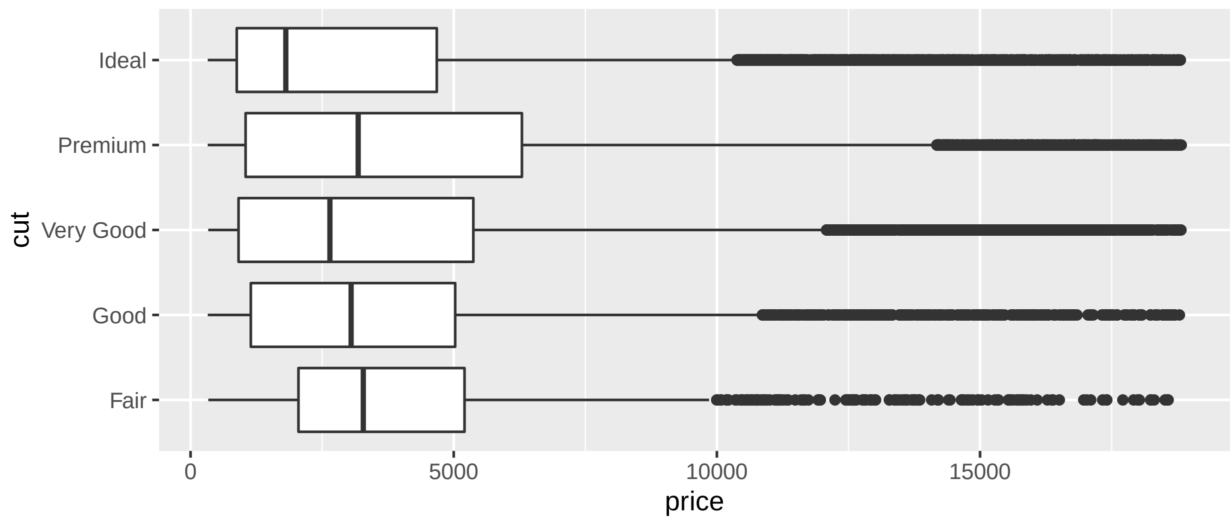


The distribution of a continuous variable can be also shown by a box plot using `geom_boxplot()`.

```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot()
```

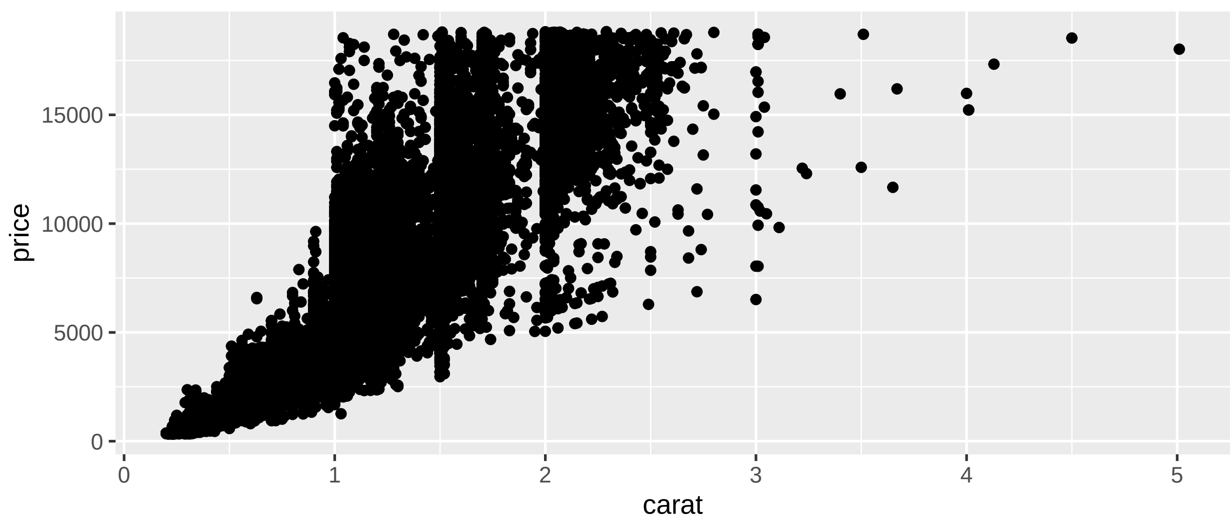



```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot()  + coord_flip()
```

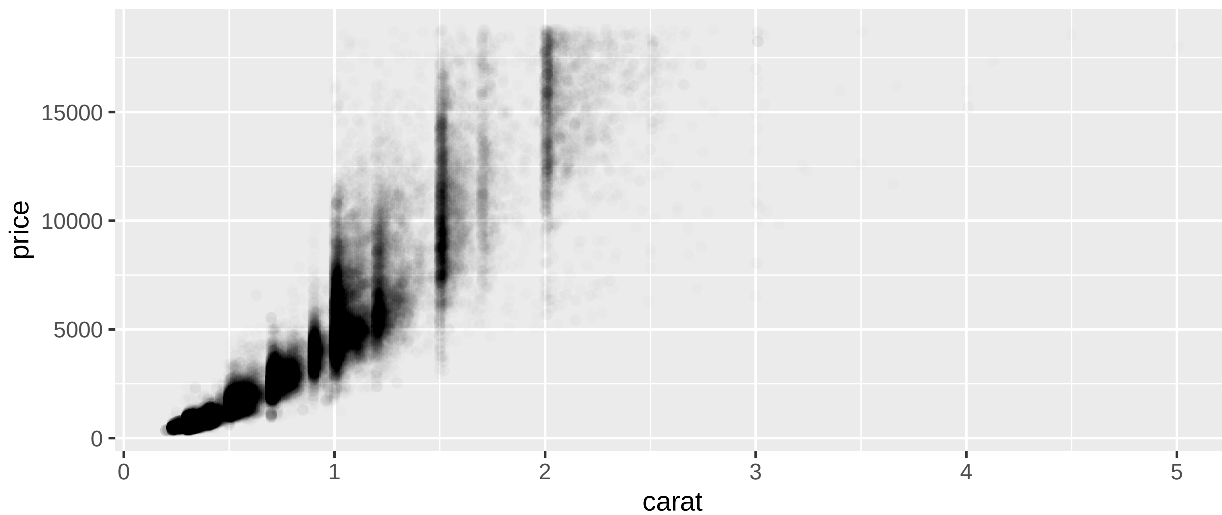


We can also show the relationship between the carat size and the price of a diamond.

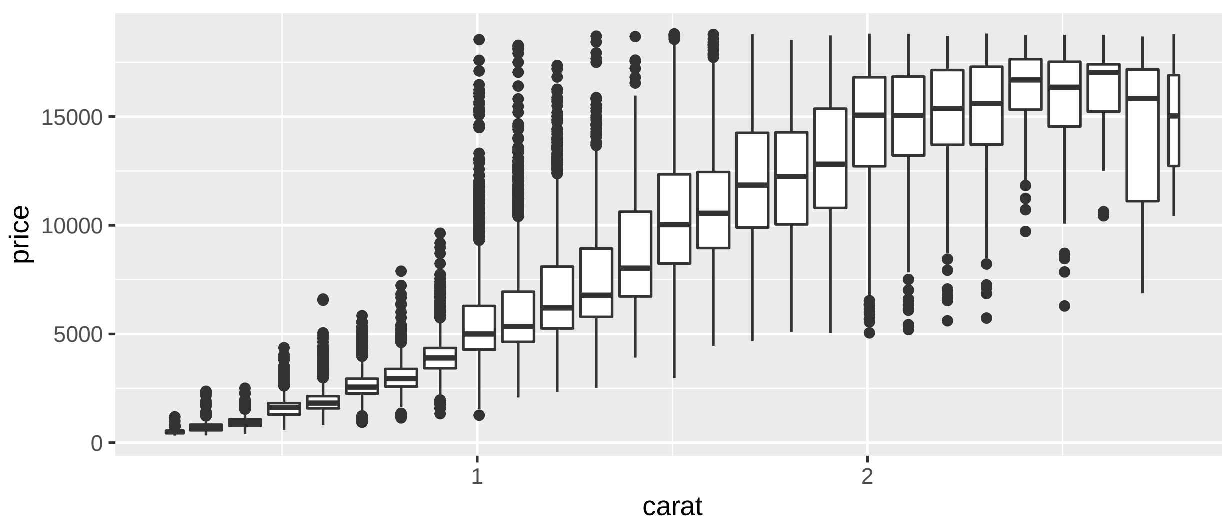
```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```




```
ggplot(data = diamonds) +
  geom_point(
    mapping = aes(x = carat, y = price),
    alpha = 1 / 100)
```



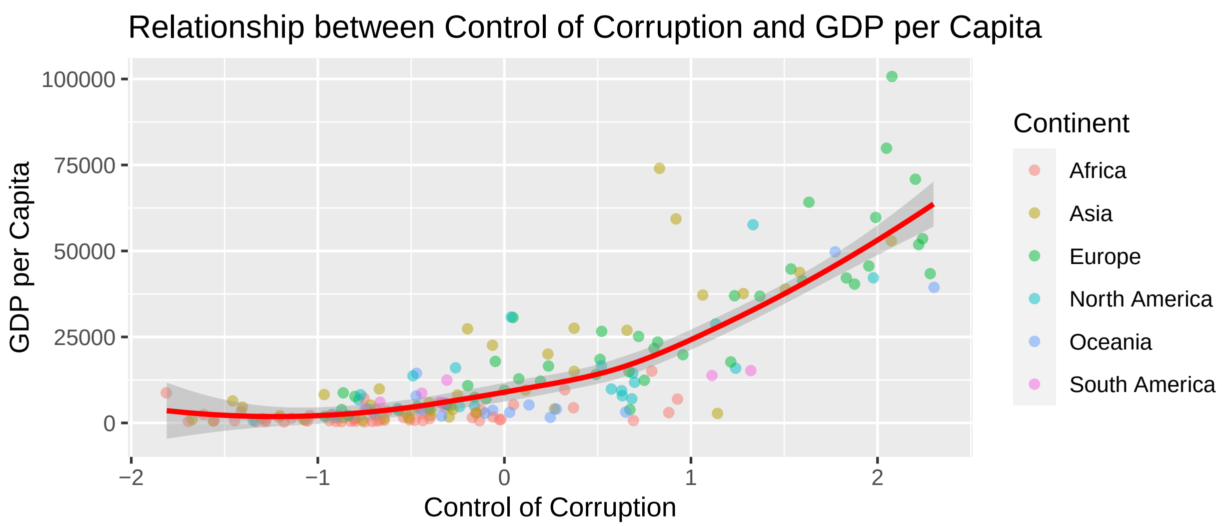

```r
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)))
```



Let's improve our `ggplot2` skills and analyze the relationship between World Bank governance indicators and GDP per capita. Let's load the Governance and GDP per Capita data set. Next, we look at the relationship between GDP per Capita and Control of Corruption. You can do the rest of governance variables as exercise.

```r
GDP_Gov <- read_csv(file = 'https://www.soybilgen.com/files/Governance_Output.csv')
head(GDP_Gov,4)
## # A tibble: 4 x 8
##   `Country Name` `Country Code` Continent `Control of Cor~ `Government Eff~
##   <chr>          <chr>          <chr>                <dbl>            <dbl>
## 1 Algeria        DZA            Africa               -0.69            -0.54
## 2 Austria        AUT            Europe                1.54             1.51
## 3 Belgium        BEL            Europe                1.6              1.33
## 4 Benin          BEN            Africa               -0.48            -0.570
## # ... with 3 more variables: `Regulatory Quality` <dbl>, `Voice and
## #   Accountability` <dbl>, `GDP Per Capita` <dbl>
plot1 <- ggplot(aes(x = `Control of Corruption`, y = `GDP Per Capita`,
                    color=Continent), data = GDP_Gov) +
```

```r
  geom_point(alpha=1/2, position = position_jitter(h=0))  +
  xlab("Control of Corruption") + ylab("GDP per Capita") +
  labs(title="Relationship between Control of Corruption and GDP per Capita") +
  geom_smooth(color='red')
plot(plot1)
```



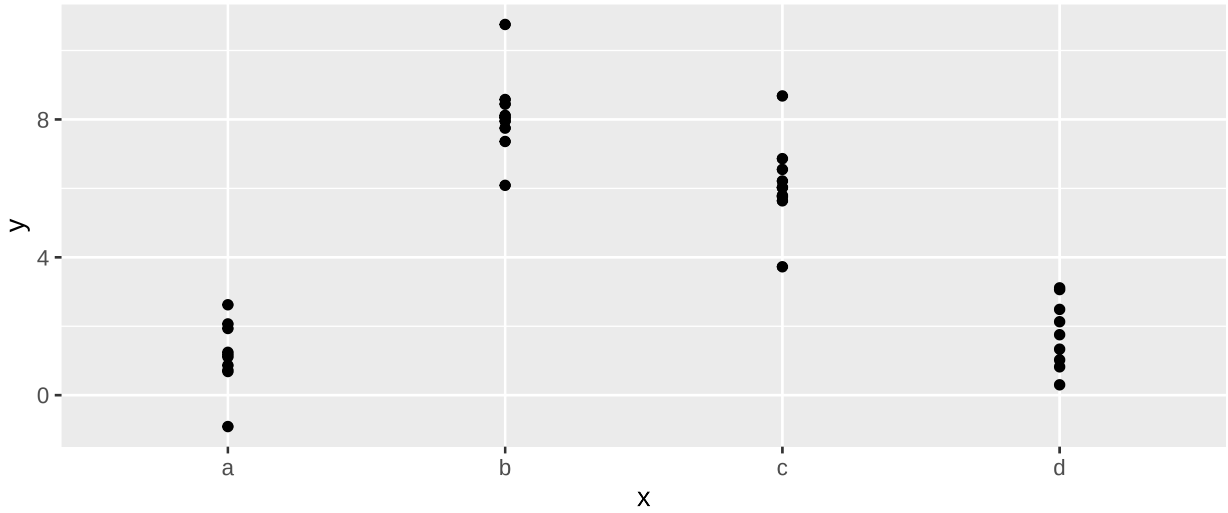
Modeling with `tidyverse`

In this part, we will use the `modelr` package. The goal of `modelr` is to provide functions that help you create elegant pipelines when modeling.

```r
require("modelr")
```

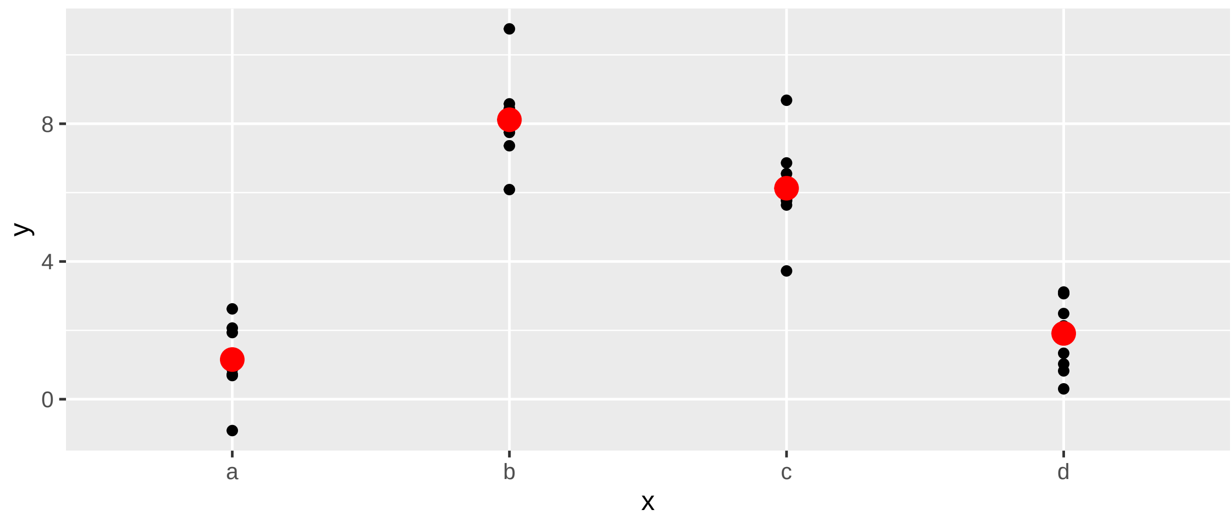
Let's start with a simple data set titled `sim2` containing one categorical and one continuous variable.

```r
ggplot(sim2) +
  geom_point(aes(x, y))
```



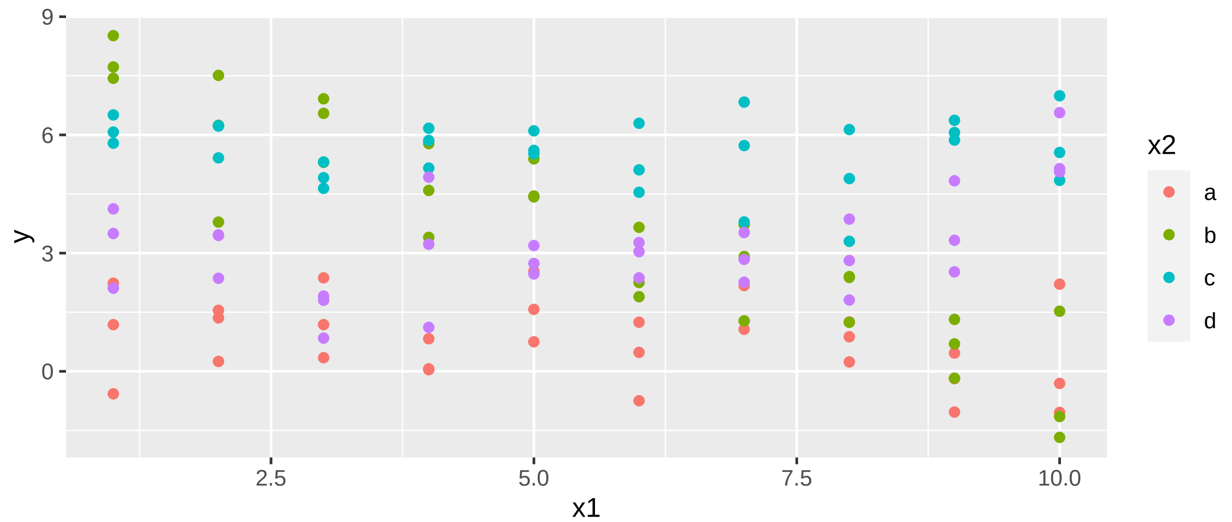
We can fit a model to it, and generate predictions.

```
mod2 <- lm(y ~ x, data = sim2)
grid <- sim2 %>%
  data_grid(x) # generate an evenly spaced grid of points from the data
str(grid)
## tibble [4 x 1] (S3: tbl_df/tbl/data.frame)
##  $ x: chr [1:4] "a" "b" "c" "d"
grid <- grid %>% add_predictions(mod2)
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) + geom_point(data = grid, aes(y = pred),
    color = "red", size = 4)
```



Next, let's work with 2 continuous variable and one categorical variable.

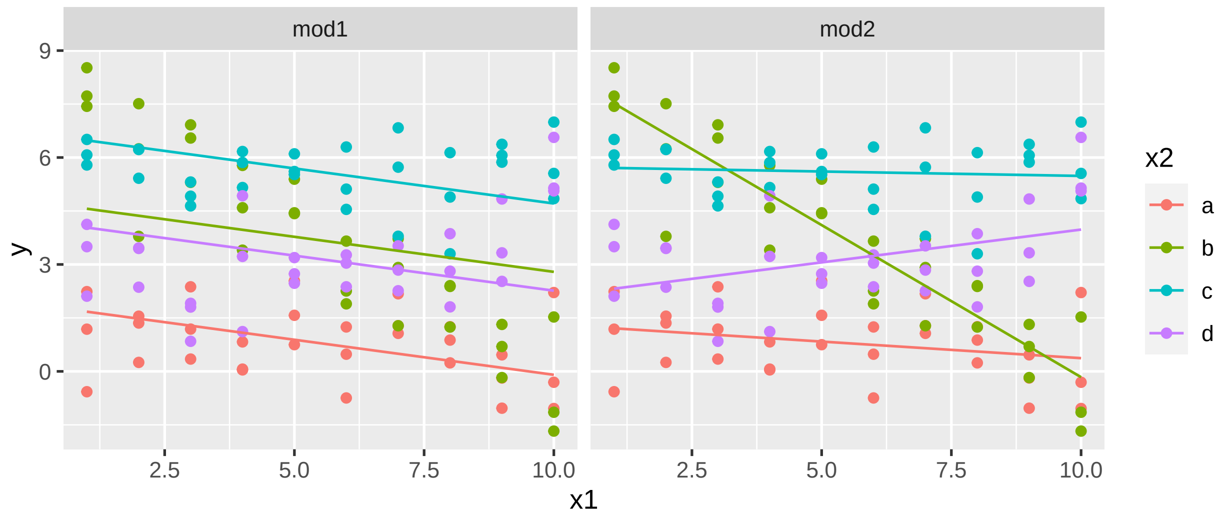
```
ggplot(sim3, aes(x1, y)) +
  geom_point(aes(color = x2))
```



We can model this with or without interaction variables. We can also visualize the results for both models on one plot using `facet_wrap()`.

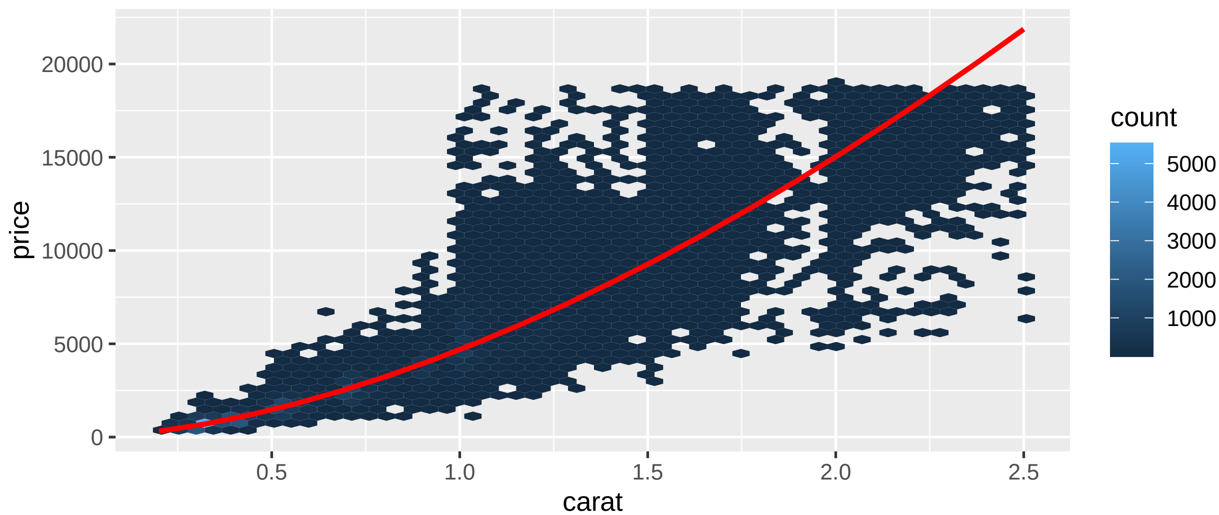
```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3) # equivalent to `y ~ x1 + x2 + x1 * x2`
```

```
grid <- sim3 %>%
  data_grid(x1, x2)
str(grid)
## tibble [40 x 2] (S3: tbl_df/tbl/data.frame)
##  $ x1: int [1:40] 1 1 1 1 2 2 2 2 3 3 ...
##  $ x2: Factor w/ 4 levels "a","b","c","d": 1 2 3 4 1 2 3 4 1 2 ...
grid <- grid %>% gather_predictions(mod1, mod2) # if we want to add each prediction
# to a new column if we can use spread_predictions()
ggplot(sim3, aes(x1, y, color = x2)) +
  geom_point() + geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```



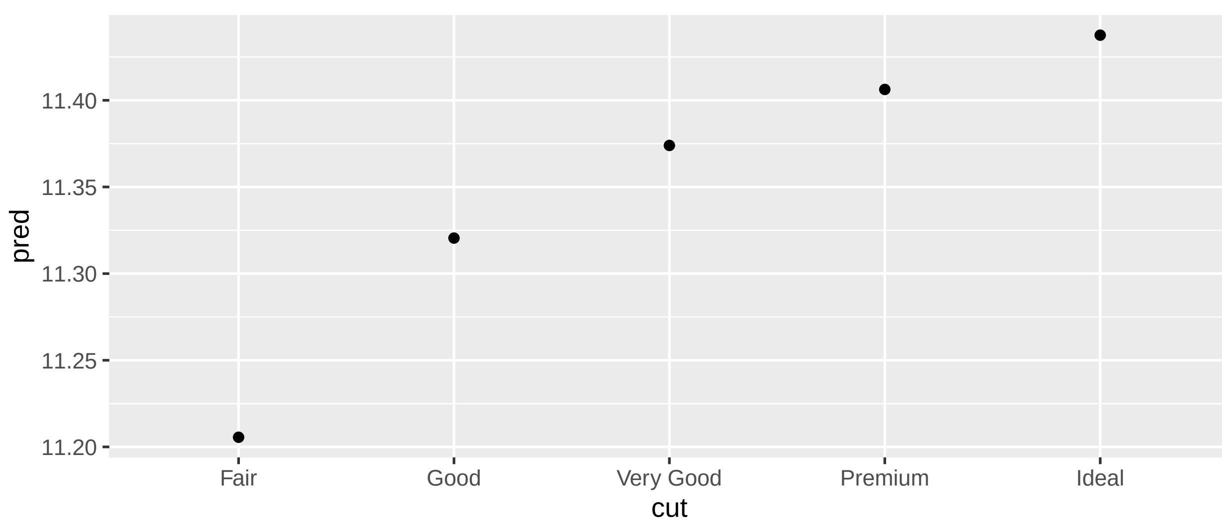
Let's use our knowledge up to this point and predict prices of diamonds using `carat`, `color`, `cut`, and `clarity`. First we start with a simple model.

```
require(hexbin) # needed for geom_hex(). geom_hex() divides the plane
# into regular hexagons, counts the number of cases in each hexagon,
# and then (by default) maps the number of cases to the hexagon fill.
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2) # start with the easy model
grid <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) # create 20 evenly separated data points
# by fixing highest and lowest values in carat
str(grid)
## tibble [20 x 1] (S3: tbl_df/tbl/data.frame)
##  $ carat: num [1:20] 0.2 0.321 0.442 0.563 0.684 ...
grid <- grid %>%
  mutate(lcarat = log2(carat)) %>%
  add_predictions(mod_diamond, "lprice") %>%
  mutate(price = 2 ^ lprice)
ggplot(diamonds2, aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, color = "red", size = 1)
```

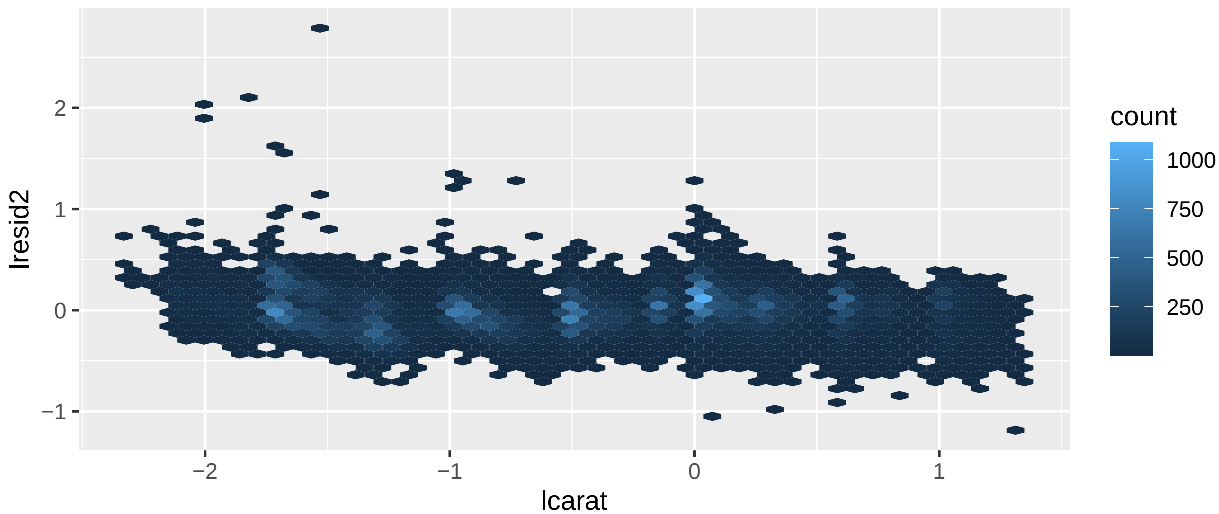


Next, we can move to more complicated model.

```
mod_diamond2 <- lm(
  lprice ~ lcarat + color + cut + clarity,
  data = diamonds2) # a more complicated model
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) # ".model =" will create "typical" values
# not supplied previously. In this case, cut is supplied, so 5 values for cut is present,
# lcarat, color, and clarity is either median if it is a continuous variable or
# the most frequent value if it is categorical value.
str(grid)
## tibble [5 x 4] (S3: tbl_df/tbl/data.frame)
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 1 2 3 4 5
##  $ lcarat : num [1:5] -0.515 -0.515 -0.515 -0.515 -0.515
##  $ color  : chr [1:5] "G" "G" "G" "G" ...
##  $ clarity: chr [1:5] "VS2" "VS2" "VS2" "VS2" ...
grid <- grid %>% add_predictions(mod_diamond2)
ggplot(grid, aes(cut, pred)) +
  geom_point()
```

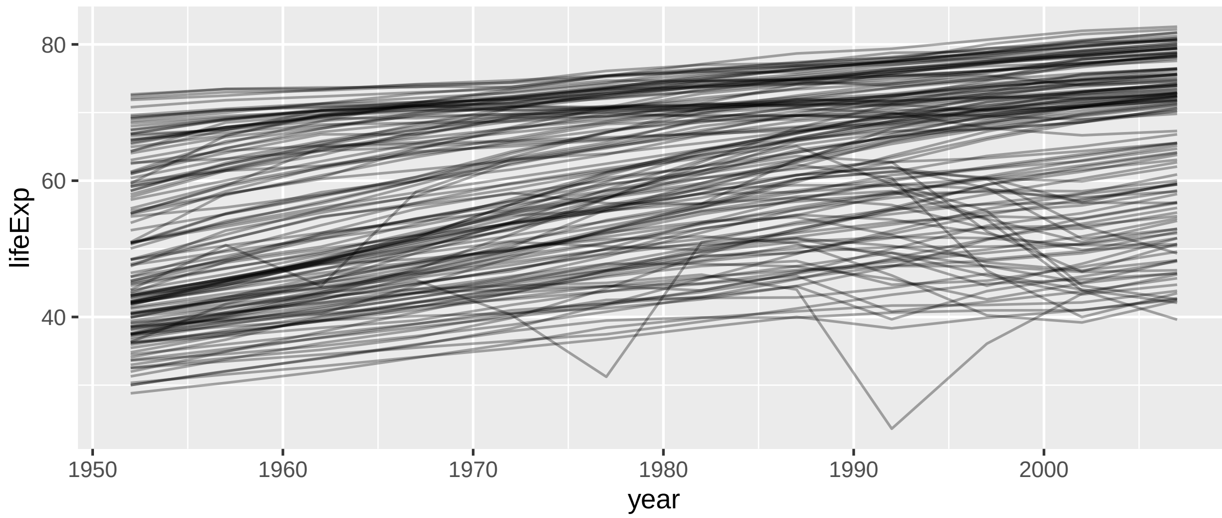


```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2, "lresid2")
ggplot(diamonds2, aes(lcarat, lresid2)) +
  geom_hex(bins = 50)
```



As we learn the basic of model fitting in `tidyverse`. Now, we will fit hundreds of models using `broom` and `purrr` packages and then compare them. `broom` tidies 100+ models from popular modeling packages and almost all of the model objects in the stats package that comes with base `R`. Let's start by loading `gapminder` data set.

```
require(gapminder)
head(gapminder,4)
## # A tibble: 4 x 6
##   country     continent  year lifeExp      pop gdpPercap
##   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
## 1 Afghanistan Asia       1952    28.8  8425333      779.
## 2 Afghanistan Asia       1957    30.3  9240934      821.
## 3 Afghanistan Asia       1962    32.0 10267083      853.
## 4 Afghanistan Asia       1967    34.0 11537966      836.
gapminder %>%
  ggplot(aes(year, lifeExp, group = country)) +
    geom_line(alpha = 1/3)
```



To create models for each country, we start by nesting data according to `country` and `continent`. We have `list` of `data frames` under the `data` column.

```
by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest()
head(by_country,4)
## # A tibble: 4 x 3
## # Groups:   country, continent [4]
##   country     continent data
##   <fct>       <fct>     <list>
## 1 Afghanistan Asia      <tibble [12 x 4]>
## 2 Albania     Europe    <tibble [12 x 4]>
## 3 Algeria     Africa    <tibble [12 x 4]>
## 4 Angola      Africa    <tibble [12 x 4]>
by_country[2,]
## # A tibble: 1 x 3
## # Groups:   country, continent [1]
##   country continent data
##   <fct>   <fct>     <list>
## 1 Albania Europe    <tibble [12 x 4]>
head(by_country$data[[2]],4)
## # A tibble: 4 x 4
##    year lifeExp     pop gdpPercap
##   <int>   <dbl>   <int>     <dbl>
## 1  1952    55.2 1282697     1601.
## 2  1957    59.3 1476505     1942.
## 3  1962    64.8 1728137     2313.
## 4  1967    66.2 1984060     2760.
by_country[10,]
## # A tibble: 1 x 3
## # Groups:   country, continent [1]
##   country continent data
##   <fct>   <fct>     <list>
## 1 Belgium Europe    <tibble [12 x 4]>
head(by_country$data[[10]],4)
## # A tibble: 4 x 4
```

```
##    year lifeExp     pop gdpPercap
##   <int>   <dbl>   <int>     <dbl>
## 1  1952    68   8730405     8343.
## 2  1957    69.2 8989111     9715.
## 3  1962    70.2 9218400    10991.
## 4  1967    70.9 9556500    13149.
```

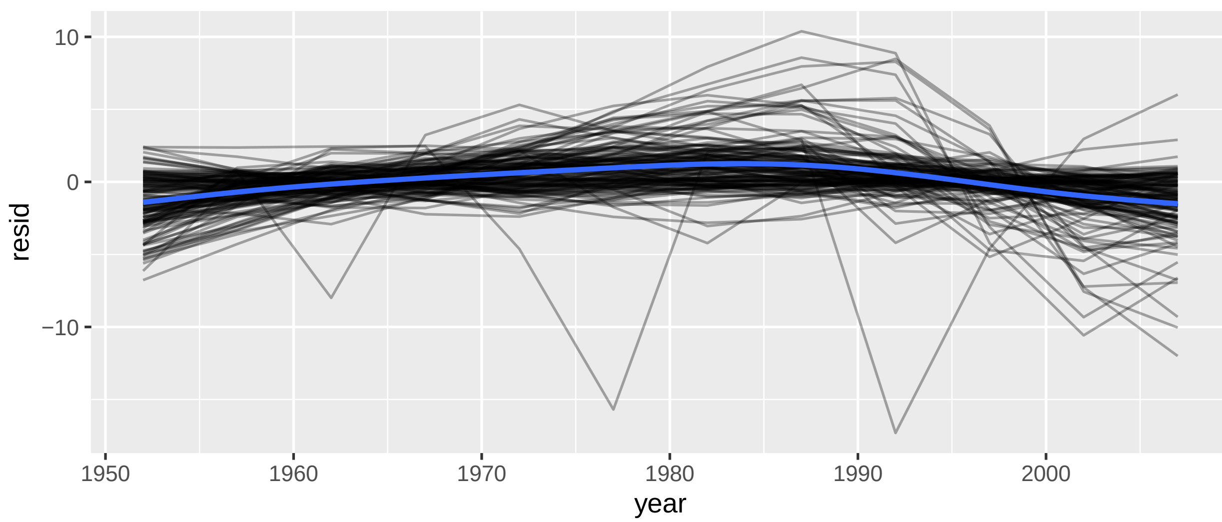
Now, we can apply same model to every country in the data set and then insert it to our `tibble`.

```
country_model <- function(df) {
  lm(lifeExp ~ year, data = df)}
by_country <- by_country %>%
  mutate(model = map(data, country_model))
head(by_country,4)
## # A tibble: 4 x 4
## # Groups:   country, continent [4]
##   country     continent data              model
##   <fct>       <fct>     <list>            <list>
## 1 Afghanistan Asia      <tibble [12 x 4]> <lm>
## 2 Albania     Europe    <tibble [12 x 4]> <lm>
## 3 Algeria     Africa    <tibble [12 x 4]> <lm>
## 4 Angola      Africa    <tibble [12 x 4]> <lm>
```

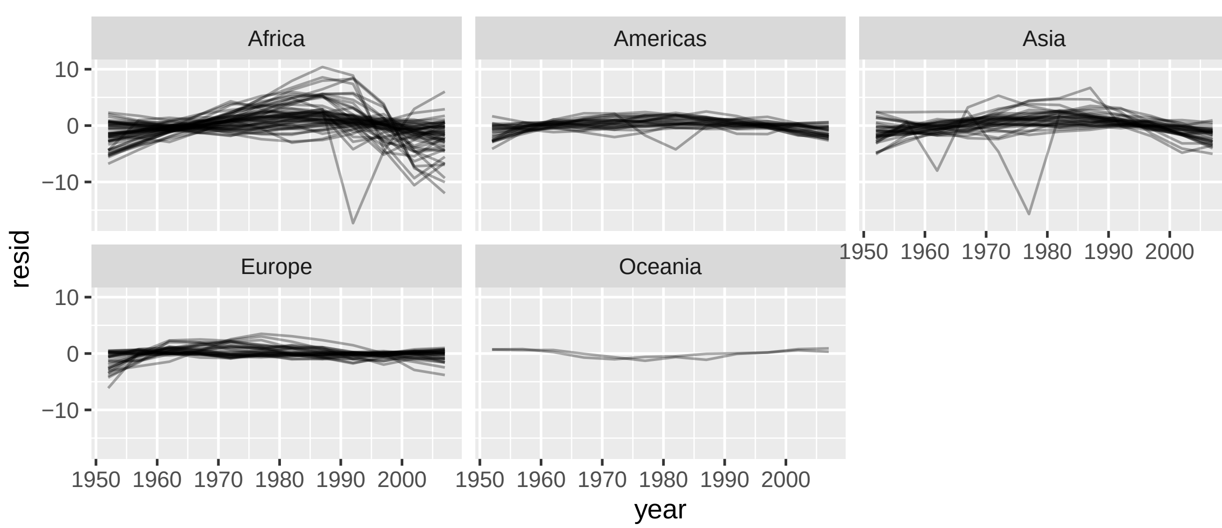
We have 142 `data frames` and 142 models. To analyze all models' residuals, we first use `add_residuals()` with each model–data pair. Then `unnest()` `residual` column to analyze all models' residuals.

```
by_country <- by_country %>%
  mutate(resids = map2(data, model, add_residuals))
head(by_country,4)
## # A tibble: 4 x 5
## # Groups:   country, continent [4]
##   country     continent data              model  resids
##   <fct>       <fct>     <list>            <list> <list>
## 1 Afghanistan Asia      <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
## 2 Albania     Europe    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
## 3 Algeria     Africa    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
## 4 Angola      Africa    <tibble [12 x 4]> <lm>   <tibble [12 x 5]>
head(by_country$resids[[1]],4)
## # A tibble: 4 x 5
##    year lifeExp      pop gdpPercap   resid
##   <int>   <dbl>    <int>     <dbl>   <dbl>
## 1  1952    28.8  8425333      779. -1.11
## 2  1957    30.3  9240934      821. -0.952
## 3  1962    32.0 10267083      853. -0.664
## 4  1967    34.0 11537966      836. -0.0172
head(by_country$data[[1]],4)
## # A tibble: 4 x 4
##    year lifeExp      pop gdpPercap
##   <int>   <dbl>    <int>     <dbl>
## 1  1952    28.8  8425333      779.
## 2  1957    30.3  9240934      821.
## 3  1962    32.0 10267083      853.
## 4  1967    34.0 11537966      836.
resids <- unnest(by_country, resids)
head(resids, 3)
```

```
## # A tibble: 3 x 9
## # Groups:   country, continent [1]
##   country     continent data          model  year lifeExp     pop gdpPercap  resid
##   <fct>       <fct>     <list>        <lis> <int>   <dbl>   <int>     <dbl>  <dbl>
## 1 Afghanist~ Asia      <tibble [12~ <lm>   1952    28.8 8.43e6      779. -1.11
## 2 Afghanist~ Asia      <tibble [12~ <lm>   1957    30.3 9.24e6      821. -0.952
## 3 Afghanist~ Asia      <tibble [12~ <lm>   1962    32.0 1.03e7      853. -0.664
resids %>%
  ggplot(aes(year, resid)) +
    geom_line(aes(group = country), alpha = 1 / 3) +
    geom_smooth(se = FALSE)
```

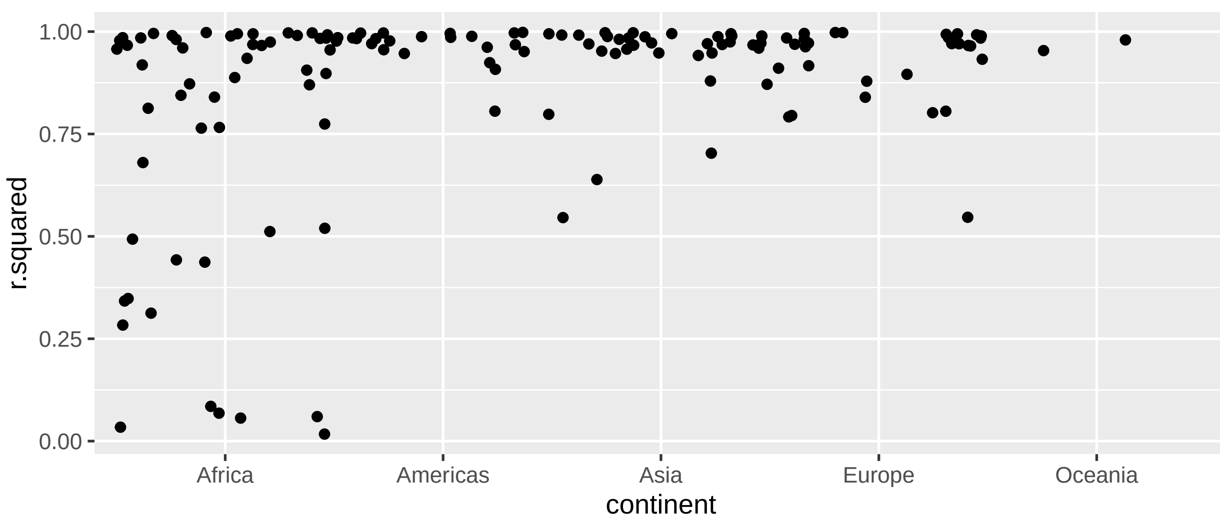



```
resids %>%
  ggplot(aes(year, resid, group = country)) +
    geom_line(alpha = 1 / 3) +
    facet_wrap(~continent)
```

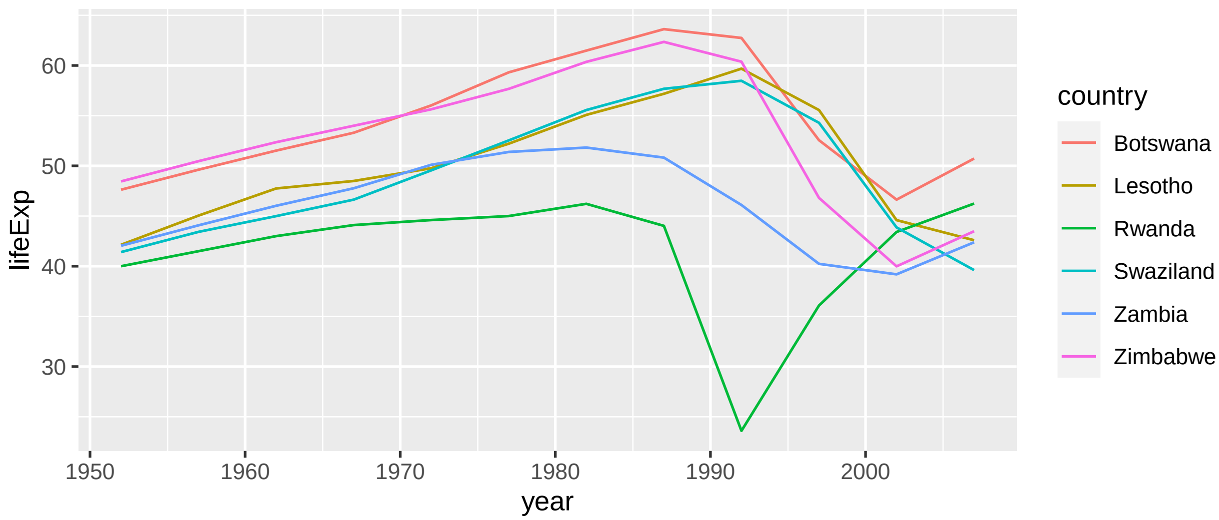


Instead of looking at the residuals from the model, we could look at some general measurements of model quality by using `broom::glance()` to extract some model quality metrics.

```r
glance <- by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance)
head(glance,4)
## # A tibble: 4 x 17
## # Groups:   country, continent [4]
##   country continent data  model resids r.squared adj.r.squared sigma statistic
##   <fct>   <fct>     <lis> <lis> <list>     <dbl>         <dbl> <dbl>     <dbl>
## 1 Afghan~ Asia      <tib~ <lm>  <tibb~     0.948         0.942  1.22     181.
## 2 Albania Europe    <tib~ <lm>  <tibb~     0.911         0.902  1.98     102.
## 3 Algeria Africa    <tib~ <lm>  <tibb~     0.985         0.984  1.32     662.
## 4 Angola  Africa    <tib~ <lm>  <tibb~     0.888         0.877  1.41      79.1
## # ... with 8 more variables: p.value <dbl>, df <dbl>, logLik <dbl>, AIC <dbl>,
## #   BIC <dbl>, deviance <dbl>, df.residual <int>, nobs <int>
glance %>%
  arrange(r.squared) %>% head(4)
## # A tibble: 4 x 17
## # Groups:   country, continent [4]
##   country continent data  model resids r.squared adj.r.squared sigma statistic
##   <fct>   <fct>     <lis> <lis> <list>     <dbl>         <dbl> <dbl>     <dbl>
## 1 Rwanda  Africa    <tib~ <lm>  <tibb~    0.0172       -0.0811  6.56     0.175
## 2 Botswa~ Africa    <tib~ <lm>  <tibb~    0.0340       -0.0626  6.11     0.352
## 3 Zimbab~ Africa    <tib~ <lm>  <tibb~    0.0562       -0.0381  7.21     0.596
## 4 Zambia  Africa    <tib~ <lm>  <tibb~    0.0598       -0.0342  4.53     0.636
## # ... with 8 more variables: p.value <dbl>, df <dbl>, logLik <dbl>, AIC <dbl>,
## #   BIC <dbl>, deviance <dbl>, df.residual <int>, nobs <int>
glance %>%
  ggplot(aes(continent, r.squared)) +
    geom_jitter(width = 0.5)
```




```r
bad_fit <- filter(glance, r.squared < 0.25)
gapminder %>%
  semi_join(bad_fit, by = "country") %>%
  ggplot(aes(year, lifeExp, color = country)) +
    geom_line()
```


60
50
40
30
lifeExp
1950
1960
1970
1980
1990
2000
year
country
Botswana
Lesotho
Rwanda
Swaziland
Zambia
Zimbabwe