Intro to R Programming

-- Nikhil Vidhani

What is a program?

- Translation of an algorithm into a language that computer understands
- An algorithm takes input, perform some operations and gives output
    - Executes in finite time
    - E.g. sorting, searching, reading, copying!

- Complexity of a Program
    - Time and space!
    - E.g. Fibonacci series!

- Programming Paradigms
    - Iterative vs Recursive
    - Procedural vs Object Oriented

- Good Program
    - Re-readable, organized and modular

Typical Programing Errors

- Syntactical (spelling mistake)
    - Will get caught very easily! Just run the program.

- Semantic Errors (meaningless operations)
    - For e.g. “nikhil”+32
    - Exceptions: like divide by 0.
        - Note that R catches this exception and returns Inf
    - May get caught. A warning will be thrown nonetheless.

- Logical Errors (Unintentional)
    - Program will crash, run forever or give a wrong answer!
    - Debugging requires some skill and experience.

Getting Help in R

- From Console
  - Just type: ? followed by function name without parenthesis
  - E.g. `?mean; ?sum; ?length;`
  - Clarify:
    - **?mean** - help for the function “mean” in the available libraries
      - Same as `help(mean)`
    - **??mean** - will perform a keyword search over the entire database
      - Same as `help.search("mean")`
    - **mean()** - call the function mean
    - **mean** - print the definition of the function “mean”

- From Web sources
  - Most reliable and easy to incorporate is www.stackoverflow.com.
  - www.r-bloggers.com is also quite helpful.
  - You can use https://cran.r-project.org for any resource on R
  - Even typing your question in google will get you good results!
    - 99% of your questions are already answered! You just need to find them!

R Input and Output

- Simple assignment
  - `X = 1;` (*or* `X <- 1;`)
  - Assignment is always right to left
    - Read 1 goes into X
    - We aren’t comparing X with 1 here
  - The semi-colon isn’t necessary in R, but it’s a good practice to use it
  - `X =  ;` is incomplete
  - # (prefix) is used as a comment. Use it for helpful comments.
  - Use Ctrl-Shift-C for multi-line comments

- Value of X can be seen by
  - `X;`

Vectors

- A sequence of numbers. Many ways to input!
  - `Y = c(1,7,-3,41); # concatenate arbitrary numbers`
  - `Y = 1:10; # natural numbers`
  - `Y = seq(1,100,9); # skip by 9`
  - `Y = rep(2, 3); # repeat 3 times`
  - `Y = rep(1:2, 3); # repeat the vector`
  - `Y = rep(1:2, each = 3); # repeat each element 3 times`
  - `Y = c(); # empty vector`
  - Execute this: `c(1:3, rep(c(5,7), each = 2), rep(9, 4), 7);`

- Length of vector: `length(Y);`

- Accessing ith element of vector: `Y[i]; # square brackets`
  - `i` should be between `1` and `length(Y)`
  - Printing the entire vector is as before: `Y;`

Objects in R

- 5 basic (atomic) types of objects
  - character – strings
  - numeric – real numbers. Also called double.
  - integer – natural numbers. Default data type for numeric vectors.
    - `typeof(1:10)`
  - complex – complex numbers. We won't use them now!
  - logical – True/False (binary)

- Most basic collection of objects is a vector (also called an array)
  - Can only contain objects of same class (i.e. character or integer; not both)
  - "list" is a special type of object and can contain heterogeneous objects
    - Any Combination of vector, matrix, atomic types etc.
    - It can even contain another list as an object. E.g. linked-lists!
    - Due to its generality its very slow and hence rarely used with large datasets unless situation demands it

Numbers

- Default type of any number is numeric (i.e. real). `typeof(1)`
- R can differentiate between corner cases:
  - `1/0` is `Inf` -- `is.infinite();`
  - `0/0` is `NaN` -- `is.nan();`
  - Missing data is `NA` -- `is.na();`
  - Check what's `Inf-Inf` ?

- Arithmetic Operations
  - `*` multiplies
  - `/` divides
  - `^` takes exponent
  - `%%` is the modulo (remainder) operator. Try: `7 %% 2;`

Coercion

- Mixing Objects
  - Automatically coerced to the same class.
  - Try: `c(1:7, "a"); c(T, 2); c("a", FALSE);`
  - Implicit coercion!
  - Never use unless you know what you’re doing!

- Explicit Coercion
  - `as.character(1:5);`
  - `as.numeric("iimb"); # warning!`
  - `as.logical(seq(-2,2,1));`

List

- Can carry different types of data together
  - `L = list(1, FALSE, 3.14, "iimb", "c", 4-3i);`
  - Print list: `L;`
  - L is in fact a list of lists. Check: `typeof(L); typeof(L[4]); typeof(L[[4]]);`
  - Single square brackets `[i]` access the ith list embedded in the list `L`
  - Double square brackets `[[i]]` access the ith element
  - Can append elements in list: `L = append(L, "7th");`
  - `unlist(L);` will coerce all elements into a single type and return a vector
  - Delete an element from a list:
    - I don't know how to do that!
    - Let's google: "delete element from list in R"
    - Open the answer on www.stackoverflow.com

Matrices

- Generalization of vectors
  - 2 dimensions instead on one!
  - N x K matrix means a matrix having N rows and K columns. Total of NK elements.
  - `M = matrix(nrow = 2, ncol = 3);`
  - Dimensions: `dim(M);`
  - Can think of M as
    - 3 columns vectors each of length 2, *or*
    - 2 row vectors each of length 3
  - Populate matrix: `M = rbind(1:3, 4:6);`
  - Alternatively populate as: `M = cbind(1:2, 3:4, 5:6);`

Matrices (cont.)

- Indexing a matrix
  - `M[i,j]` gives the element at ith row and jth column
  - `M[i,]` gives the entire ith row (a vector)
  - `M[,j]` gives the entire jth column (a vector)

- Matrix multiplication
  - `*` just does an element wise multiplication, i.e. $(M * N)_{ij} = M_{ij} * N_{ij}$
  - `%*%` performs the usual matrix multiplication. Try: `M %*% N`
    - Dimensions must match
    - Try `t(M) %*% M;`
    - `t(M)` takes transpose of a matrix!

Matrices (cont.)

- Identity matrix: `diag(3)`
- Diagonal Matrix: `diag(c(1,5,7)); diag(1:7);`
- Diagonal of a matrix: `diag(M)`
- Trace of a matrix: `sum(diag(M))`
- Inverse of a matrix:
  - Must be a square matrix: `M = matrix(1:9, nrow = 3, ncol = 3);`
    - Another way to create a matrix. Data is entered column-wise.
  - Determinant must be non-zero: `det(M); M[3,3] = 19; det(M);`
  - Inverse: `solve(M);`

Reading Data

- Download some stock data from NSE
  - https://www.nseindia.com/products/content/equities/indices/historical_index_data.htm
  - Save the CSV file as data.csv
- From CSV (most common)
  - `setwd(“C:/Users/nikhil/data.csv"); nifty = read.csv("data.csv");`
  - Alternatively: `nifty = read.csv(" C:/Users/nikhil/data.csv");`
- From Excel
  - Search it yourself! It is not recommended btw.
- From clipboard
  - `read.table("clipboard");`
  - This is quick fix for small data transfer between R and excel. Use `read.csv()` as your primary method for data reading!

Reading Data (cont.)

- Viewing data
  - `View(nifty);`
- Date
  - `nifty$Date = as.Date(nifty$Date, format = "%d-%b-%Y");`
  - `n = nrow(nifty);`
  - `d = nifty$Date[1];`
  - `format(d, format = "%D");         #  04/02/18`
  - `format(d, format = "%d-%m-%y");   #  02-04-18`
  - `format(d, format = "%d.%b.%Y");   #  02.Apr.2018`
  - `format(d, format = "%d_%B_%Y");   #  02_April_2018`
- Alternatively,
  - `read.table("data.csv", header = T, sep = ",", nrows = 5);`

if-else

-
  ```
  if(<COND_1>) {
    # do something!
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else {
    # ...
  }
  ```

-
  ```
  if(<COND_1>) {
    # do something!
  } else if(<COND_2>) {
    # ...
  } else {
    # ...
  }
  ```

```
if(nifty$Close[2] > nifty$Close[1]) {
  str = paste("Stock market closed green on", nifty$Date[2]);
} else if(nifty$Close[2] > nifty$Open[2]) {
  str = paste("Stock market closed above opening on", nifty$Date[2]);
} else {
  str = paste("Stock market was red and closed below opening on", nifty$Date[2]);
}
print(str);
```

for loop

- Looping is used to perform similar set of tasks repetitively

  -
    ```
    for(i in n:1) {
      print(nifty$Date[i]);
    }
    ```
  - `n:1;` is same as `seq(n,1,1);` i.e. backwards counting!
  - Alternatively, you can execute: `rev(nifty$Date);` or `nifty$Date[n:1];`
- Try avoiding loops if you can!
  - Increasing all dates by a week: `nifty$Date + 7`
  - Finding Daily growth: `nifty$Close[-1] / nifty$Close[-n]`
  - Daily diff. b/w high and low prices: `nifty$High - nifty$Low`
  - Question: find % growth in daily volatility
    - Volatility is defined as: $Vol_t = (High_t - Low_t)/Open_t$
    - Percentage Growth is defined as: $\%G = \frac{(Value_{t+1} - Value_t)}{Value_t} * 100$
- Loops and if-else can be used in conjunction!

Jumping

- Till now all our commands executed sequentially
- There may be circumstances when we need to jump
- Next and Break
  - `next` is used to skip an iteration, while `break` exits the loop entirely.

  -
    ```
    for(i in 1:10) {
      if(i <= 3) {
        next;
      }
      if(i > 6) {
        break;
      }
      print(i);
    }
    i;
    ```
- `return()` is used to exit a function with a value.

Function

- Organize often-used set of instructions separately in a “function”
- Calling a function will execute all the commands in the body of function
- We have used many functions till now
  - They end with parenthesis: `()`
  - E.g. `sum(); rbind(); vector(); format(); read.csv();` etc
  - Note that curly braces `{}` are used for if-else and for loops, square braces `[]` for vector/matrix indexing and parenthesis `()` for grouping, if-else & for conditions and functions.
- A function has
  - A name by which we call them, e.g. `sum`
  - A set of inputs to be put within parenthesis like numbers `1:10` in `sum()`
  - Return value which is the output of the function like the sum of numbers in `sum()`

Function Example

```
my_mean = function(x) {
  n = length(x);
  mean = sum(x) / n;
  return(mean);
}
```

- Name of the function is: `my_mean`
- Input is: `x`
- Output is: `mean`
  - Note that the mean here is just a name, we could well have used any other name without changing anything about our function

Function (Example) Cont.

- Alternate ways to write the same function

  -
    ```
    my_mean = function(x) {
      return( sum(x) / length(x) );
    }
    ```
    - No need to store sum and length. We can directly divide them!

  -
    ```
    my_mean = function(x) {
      sum(x) / length(x);
    }
    ```
    - No need for an explicit return. The last statement is returned by default.

- Try various value with `my_mean()` and the inbuilt `mean()`. See that the answers are exactly the same.

- Write a function for variance where $Var(x) = mean([x - mean(x)]^2)$
  - Compare it with the inbuilt `var()` function?

Multiple conditions & which() function

- The arguments to `if()` and `which()` and the output of `is.xx()` family of functions is a logical object, i.e. either `TRUE` or `FALSE`.

- A valid combination of logical objects is also a logical object. E.g.
  - Logical AND: `TRUE & FALSE` is `FALSE`
  - Logical OR: `TRUE | FALSE` is `TRUE`
  - Logical NOT: `! FALSE` is `TRUE`

- De Morgan’s Law
  - $!\,(A \mid B) = (!\,A)\ \&\ (!\,B)$
  - $!\,(A\ \&\ B) = (!\,A) \mid (!\,B)$

- The below two indexes are one and same (by De Morgan Law),
  - `day = as.numeric( substr(nifty$Date, 9, 10) );`
    - OR `day = as.numeric( format(df[,1], format = "%d") );`
  - `idx = which(nifty$Close > nifty$Open & (day < 5));`
  - `idx = which( !(nifty$Close < nifty$Open | (day >= 5)) );`

- `which()` gives the indexes matching the criterion. E.g. out of `101:200` which numbers are multiples of 2,3 and 5 ?
  - `count = 101:200;`
  - `which( count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0 );`
  - `count[count %% 2 == 0 & count %% 3 == 0 & count %% 5 == 0];`

- We can do multi-way match using `%in%`
  - `mult_17 = seq(17,300,17);`
  - `which(count %in% mult_17);`
  - `which(mult_17 %in% count);`
  - `which(!(count %in% mult_17));`
  - `which(!(mult_17 %in% count));`

Correlation

- Difference between Independence and Correlation
  - $X \sim N(0,1), Y = X^2$. Are X and Y correlated?
  - Let’s check in R

    -
      ```
      X = rnorm(1000, 0, 1);
      Y = X^2;
      cor(X,Y);
      ```
    - However, `cor(X,X^3)` != 0.
  - Let $Y \sim N(0,1)$ independent of X. Then is cor(X,Y) = 0? What about cor(X^q, Y)?

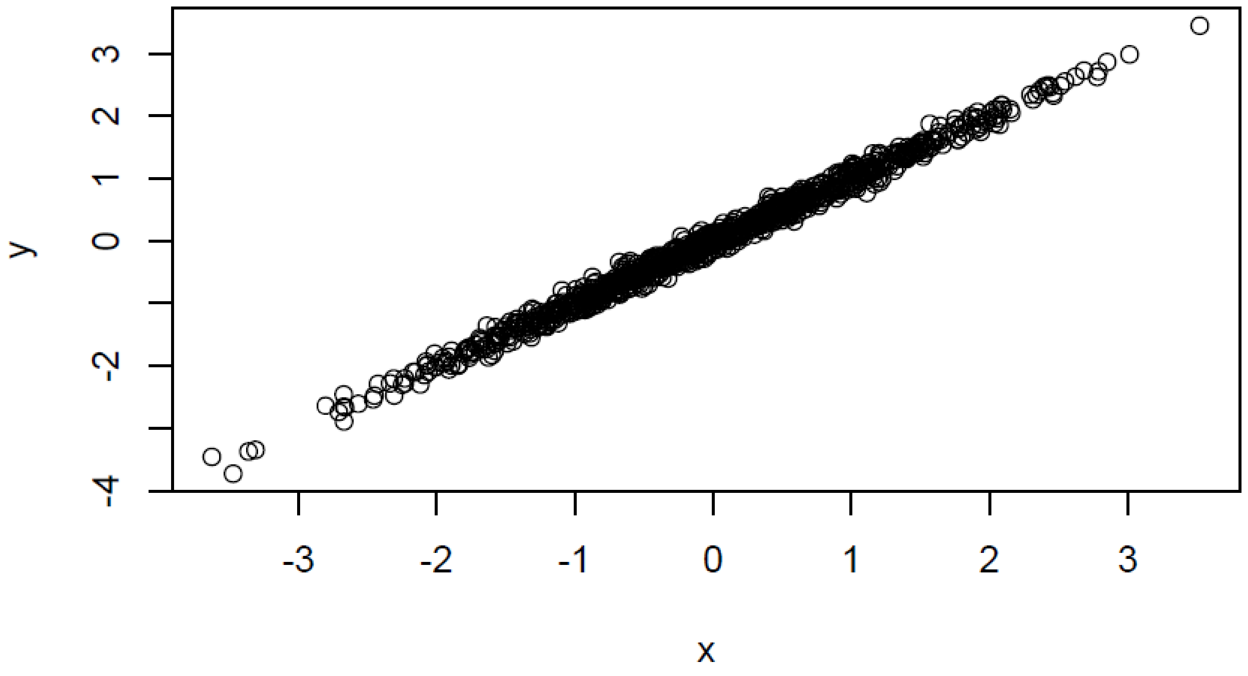
    -
      ```
      Y = rnorm(1000,0,1);
      ```

    -
      ```
      cor_q = rep(NA, 10);
      for(q in 1:10) {
          cor_q[q] = cor(X^q, Y);
      }
      cor_q;
      ```
  - Independence implies NO correlation of any functional form.

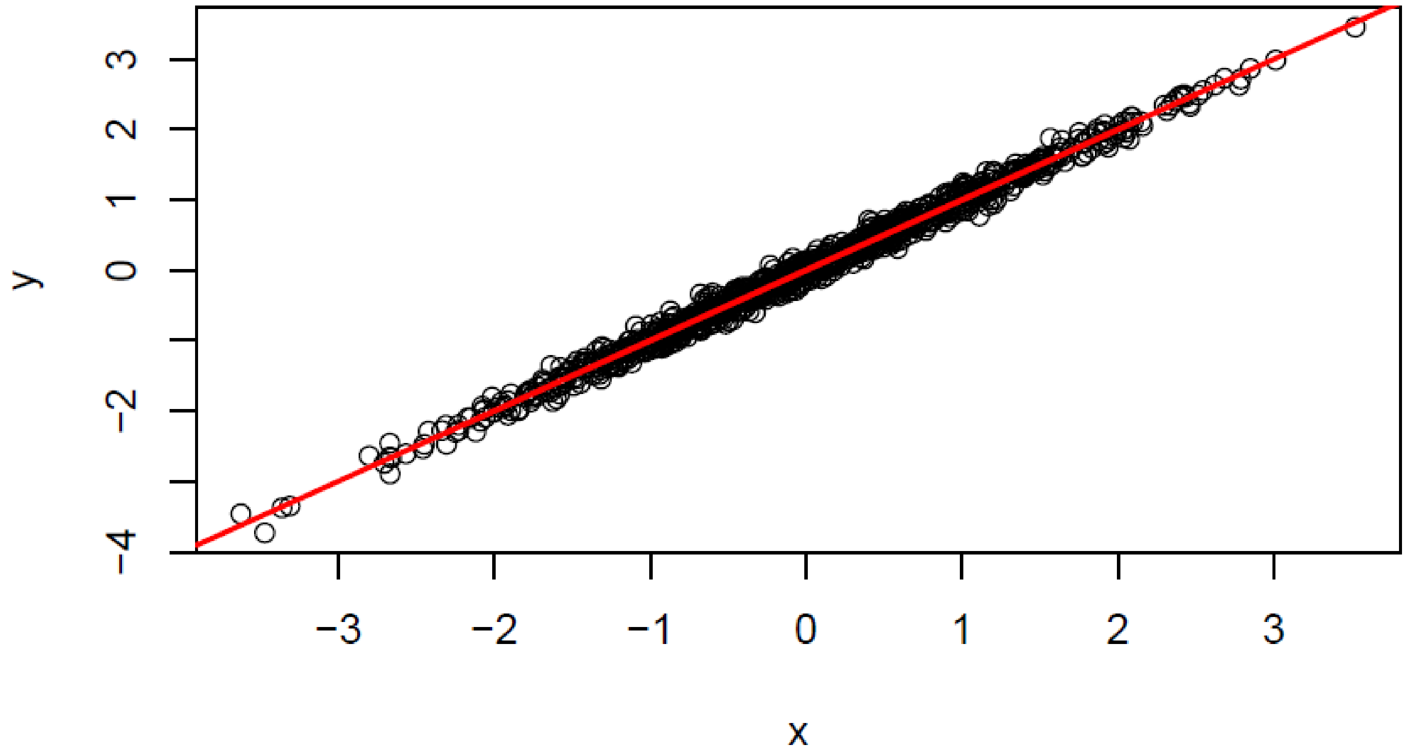
Regression

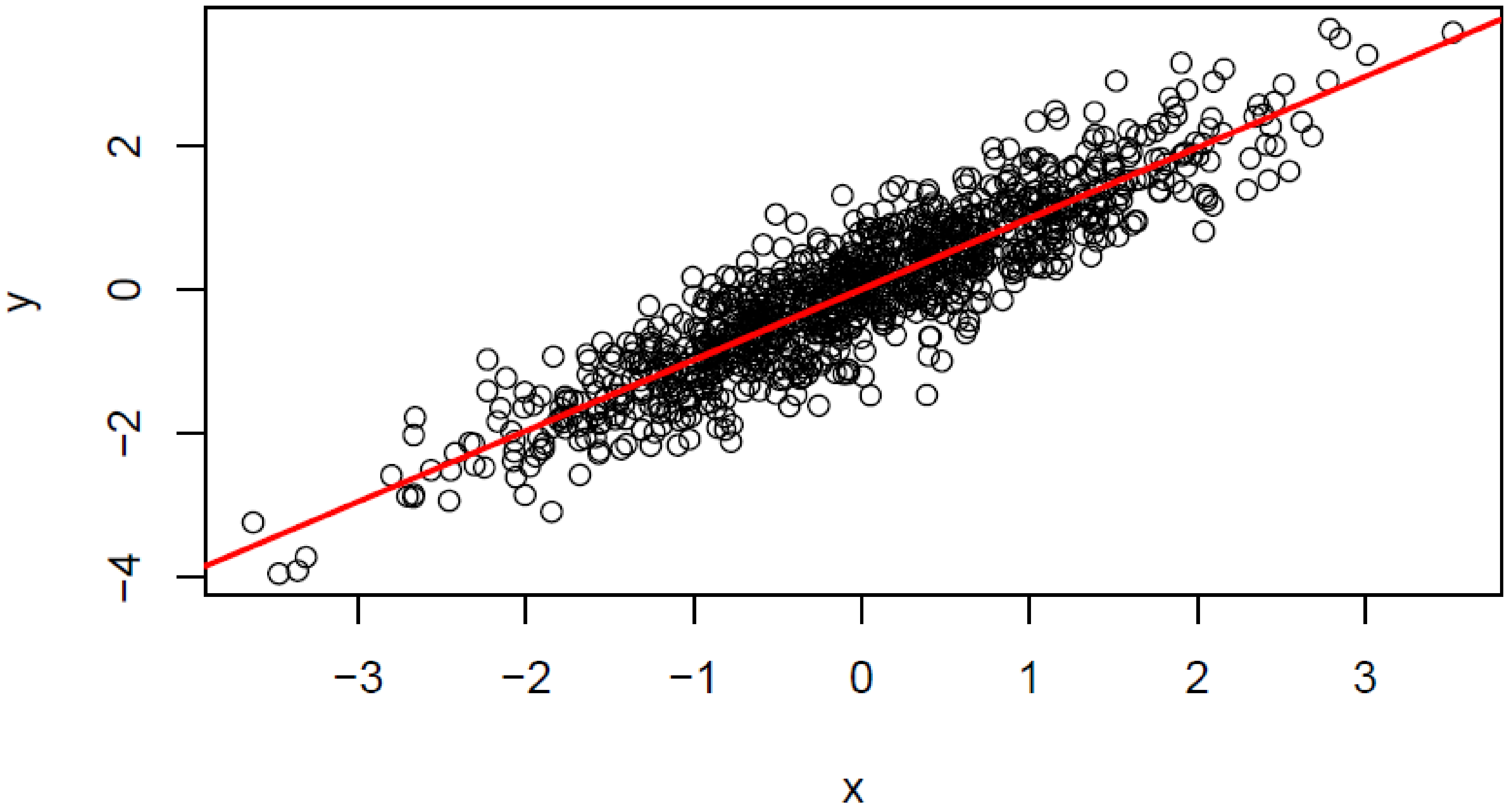
- Let, $Y = \beta_0 + \beta_1 \cdot X + u$ be the true model. By regressing Y on X, we hope to recover an unbiased estimate of β_1 and see how much of the variation in Y is explained by variation in X unrelated to variation in u.

```
n = 1000;
x = rnorm(n, 0, 1);
u = rnorm(n, 0, 0.1);
y = u + x;
plot(x,y);
```

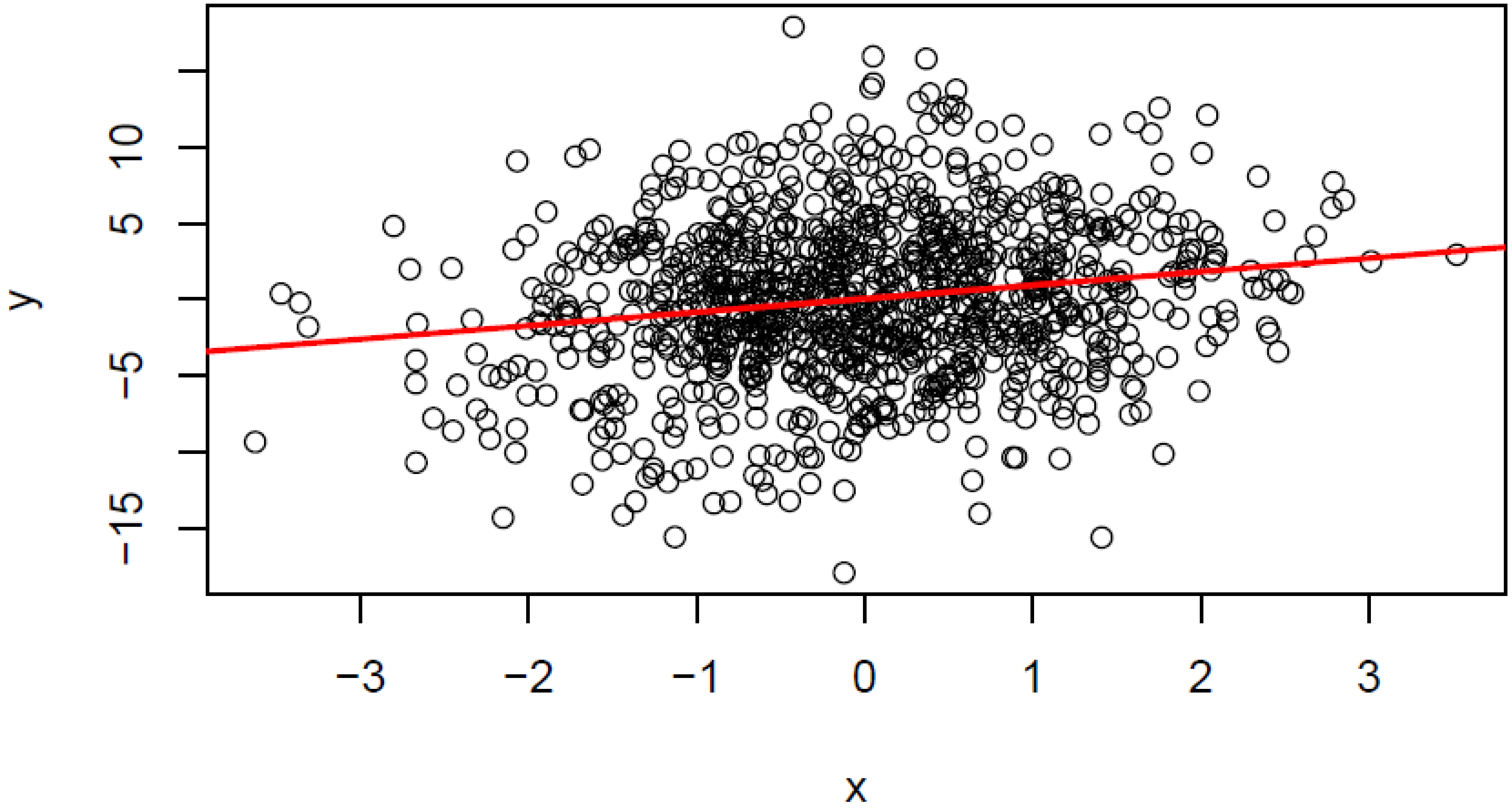


```
fit = lm(y ~ x);
summary(fit);
stargazer(fit, type = "html", out = "fit.html");
# Regression Line
abline(fit$coefficients, col = "red", lwd = 2);
```





```
u = rnorm(n, 0, 0.5);
y = u + x;
plot(x,y);
fit = lm(y ~ x);
summary(fit);
# Regression Line
abline(fit$coefficients, col = "red", lwd = 2);
```



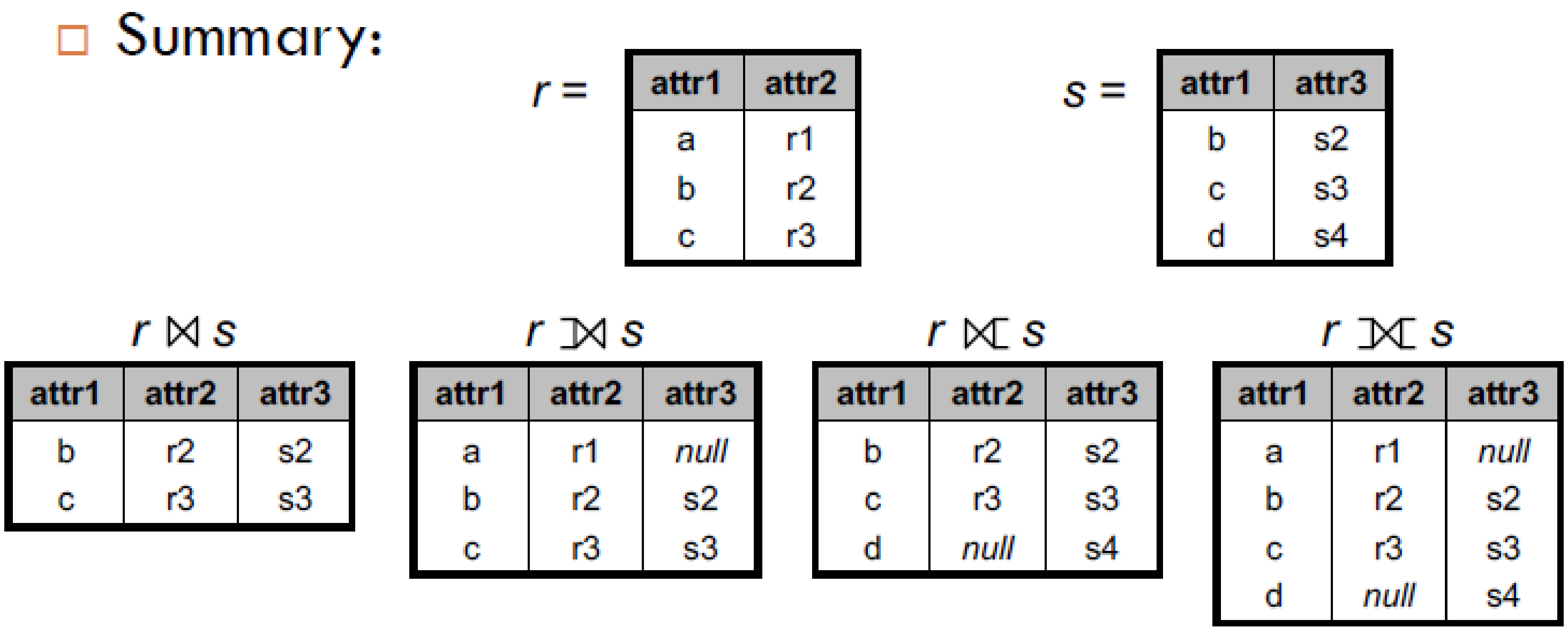
```
u = rnorm(n, 0, 5);
y = u + x;
plot(x,y);
fit = lm(y ~ x);
summary(fit);
# Regression Line
abline(fit$coefficients, col = "red", lwd = 2);
```

Some Useful functions

- `unique(), duplicated()`
- `list.files()`
    - Pattern matching using regex
    - All files starting from “s”: `"^s"`
    - All files starting with “b” or “d”: `"^(b|d)“`
    - All CSV files: `".*.csv"`
- `order()`
    - Sort data/dataframes
    - It gives the sequence of ordered indexes NOT the ordered numbers
    - Can do 2-way and 3-way sorts
- `union(), intersect()`
- `cumsum(), cumprod()`
    - `cumprod()` using only `cumsum()` ?

Merging

- Combining two datasets is a very routine and important task
- Merging is akin to Joining (in relational database, SQL etc)



□ Summary:

r =

attr1	attr2
a	r1
b	r2
c	r3

s =

attr1	attr3
b	s2
c	s3
d	s4

r ⋈ *s*

attr1	attr2	attr3
b	r2	s2
c	r3	s3

r ⟕ *s*

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3

r ⟖ *s*

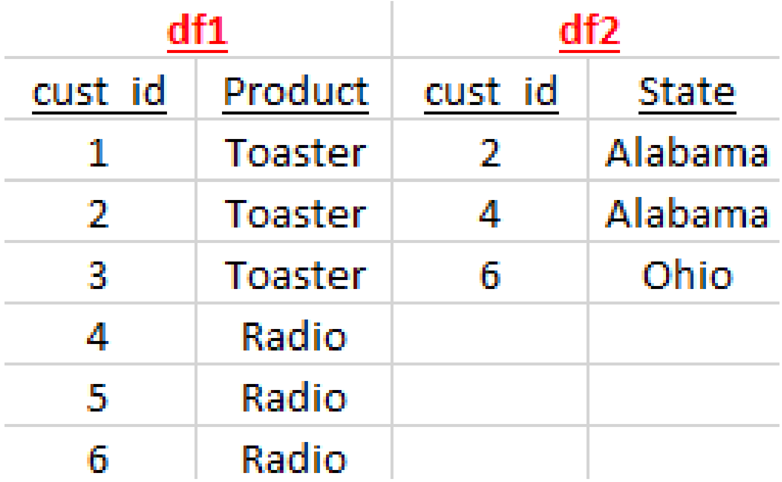
attr1	attr2	attr3
b	r2	s2
c	r3	s3
d	*null*	s4

r ⟗ *s*

attr1	attr2	attr3
a	r1	*null*
b	r2	s2
c	r3	s3
d	*null*	s4

Merging Example

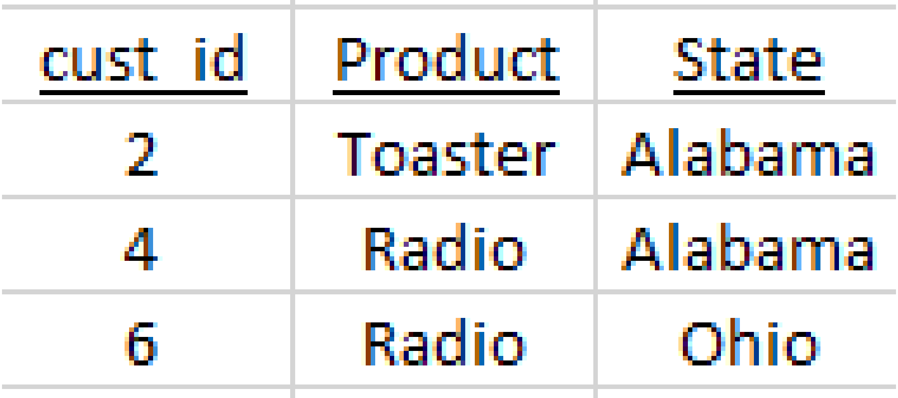
- Taken from Stackoverflow webpage
- Create two datasets:
  - `df1 = data.frame(cust_id = c(1:6), Product = c(rep("Toaster", 3), rep("Radio", 3)));`
  - `df2 = data.frame(cust_id = c(2, 4, 6), State = c(rep("Alabama", 2), rep("Ohio", 1)));`
  - The data looks like:
- We will need to use a new package `dplyr`
  - `install.packages("dplyr");`



df1		df2	
cust_id	Product	cust_id	State
1	Toaster	2	Alabama
2	Toaster	4	Alabama
3	Toaster	6	Ohio
4	Radio		
5	Radio		
6	Radio		

Inner Join

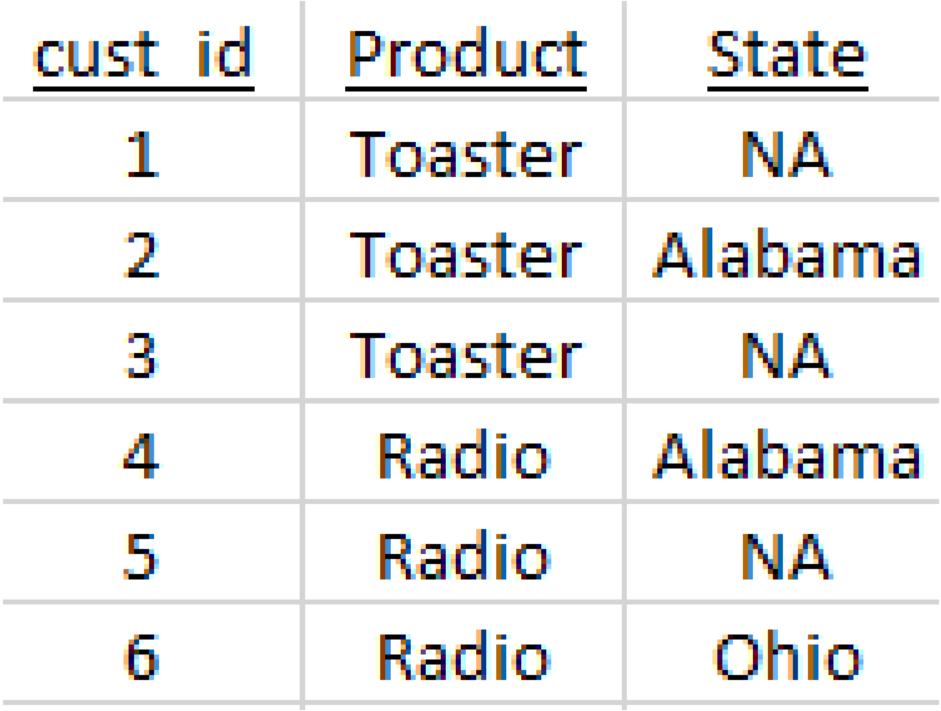
- `dplyr::inner_join(df1, df2);`



cust_id	Product	State
2	Toaster	Alabama
4	Radio	Alabama
6	Radio	Ohio

Full Join

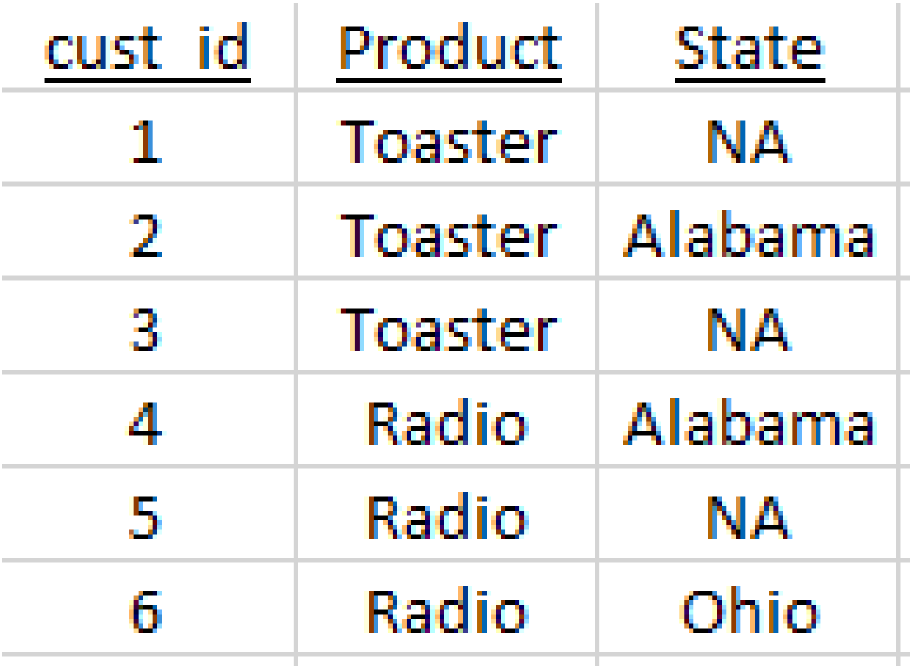
- `dplyr::full_join(df1, df2);`



cust_id	Product	State
1	Toaster	NA
2	Toaster	Alabama
3	Toaster	NA
4	Radio	Alabama
5	Radio	NA
6	Radio	Ohio

Left Join

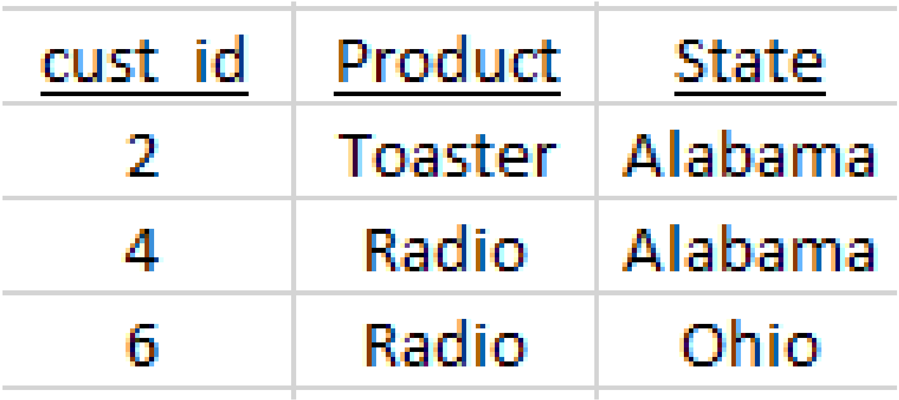
- `dplyr::left_join(df1, df2);`



cust_id	Product	State
1	Toaster	NA
2	Toaster	Alabama
3	Toaster	NA
4	Radio	Alabama
5	Radio	NA
6	Radio	Ohio

Right Join

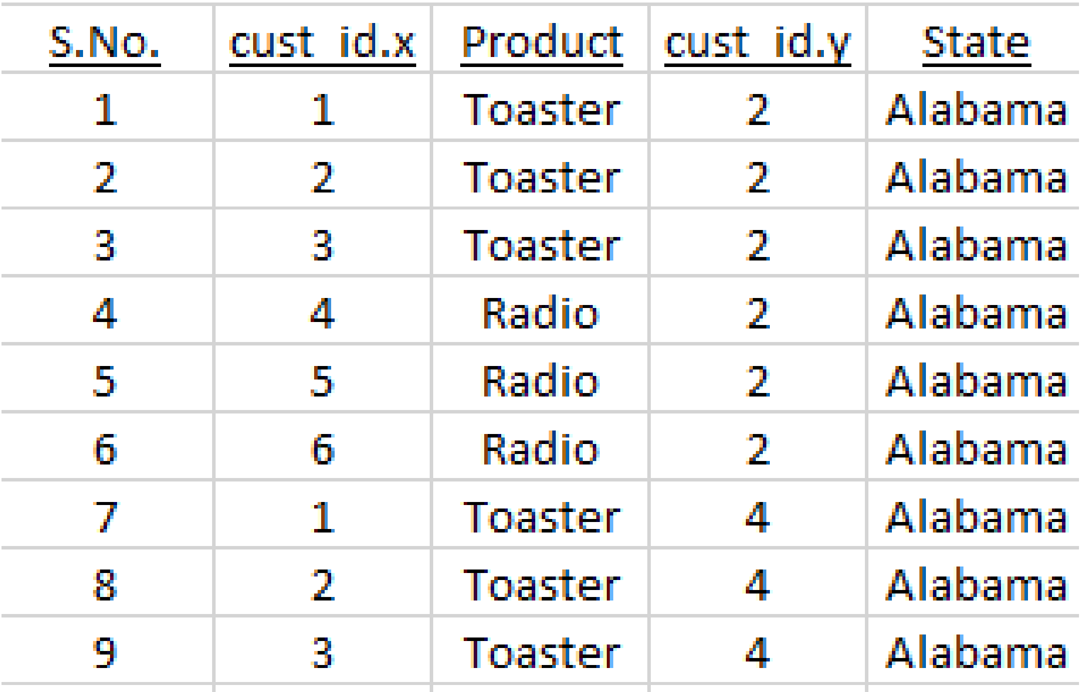
- `dplyr::right_join(df1, df2);`
  - The above is identical to, `dplyr::left_join(df2, df1);`



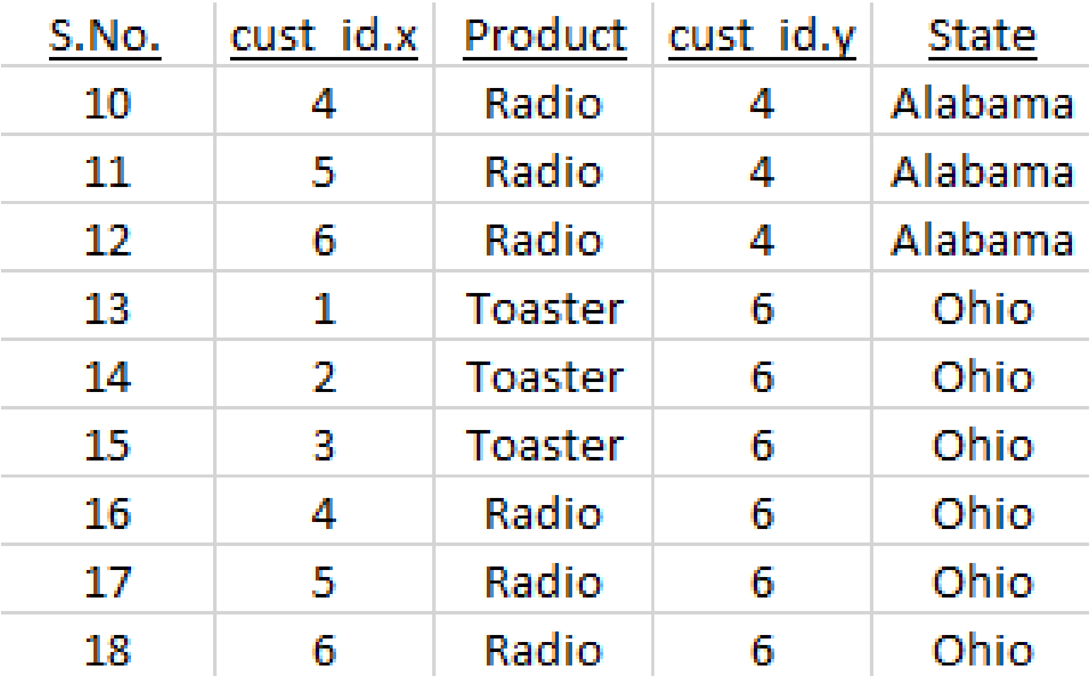
cust_id	Product	State
2	Toaster	Alabama
4	Radio	Alabama
6	Radio	Ohio

Cartesian Product

- Every row of `df1` multiplied with every row of `df2`
- `cart_prd = merge(df1, df2, by = NULL);`



S.No.	cust_id.x	Product	cust_id.y	State
1	1	Toaster	2	Alabama
2	2	Toaster	2	Alabama
3	3	Toaster	2	Alabama
4	4	Radio	2	Alabama
5	5	Radio	2	Alabama
6	6	Radio	2	Alabama
7	1	Toaster	4	Alabama
8	2	Toaster	4	Alabama
9	3	Toaster	4	Alabama



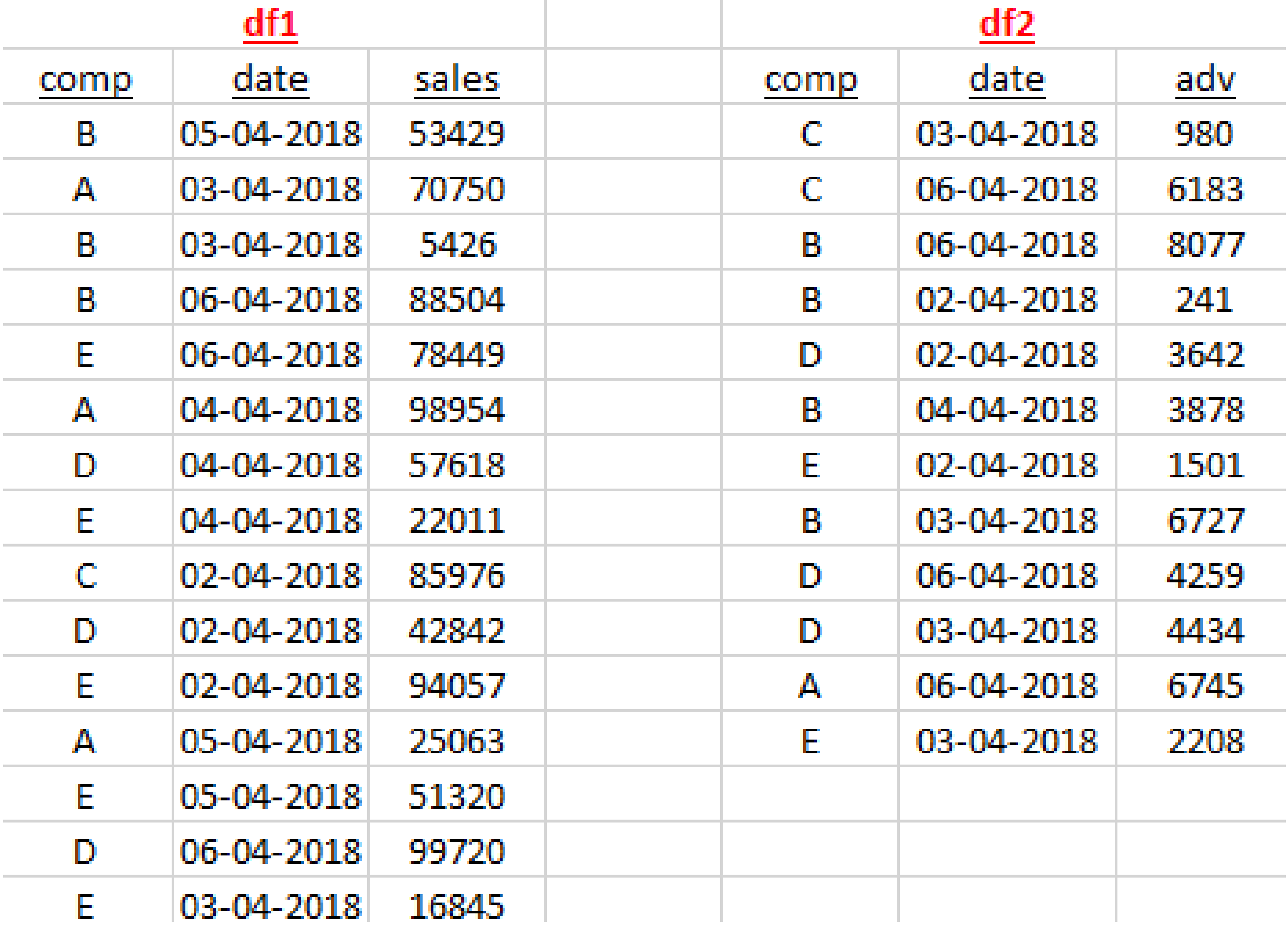
S.No.	cust_id.x	Product	cust_id.y	State
10	4	Radio	4	Alabama
11	5	Radio	4	Alabama
12	6	Radio	4	Alabama
13	1	Toaster	6	Ohio
14	2	Toaster	6	Ohio
15	3	Toaster	6	Ohio
16	4	Radio	6	Ohio
17	5	Radio	6	Ohio
18	6	Radio	6	Ohio

- Cartesian products are extremely slow. Never do that even on a decent sized (> 1000 rows) dataset. Your computer will probably hang.
- Although there is no use of Cartesian products, it encapsulates all types of merges. Meaning we can extract any type of merge from a Cartesian product.
- Inner join can be extracted via
  - `idx = which(cart_prd$cust_id.x == cart_prd$cust_id.y);`
  - `cart_prd[idx,];`

Merging with more than one variable

- Most of the merging usage is with two variables: date and company name. Generate data using below:

```
date = seq.Date(as.Date("2018-01-01"), by = 1, length.out = 5);

comp = c("A", "B", "C", "D", "E");

all_pairs = merge(comp, date, by = NULL);

# sales data - 15 points

idx = sample(1:nrow(all_pairs), 15, replace = F);

df1 = data.frame(comp = all_pairs$x[idx], date = all_pairs$y[idx],

                 sales = round(runif(15, min = 1e3, max = 1e5)));

# advertising data - 12 points

idx = sample(1:nrow(all_pairs), 12, replace = F);

df2 = data.frame(comp = all_pairs$x[idx], date = all_pairs$y[idx],

                 adv = round(runif(12, min = 1e2, max = 1e4)));
```



df1

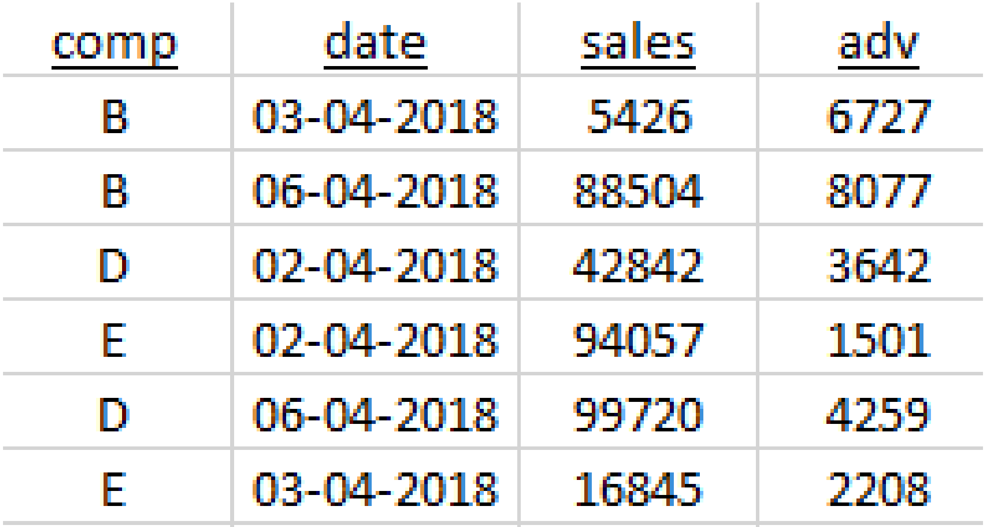
comp	date	sales
B	05-04-2018	53429
A	03-04-2018	70750
B	03-04-2018	5426
B	06-04-2018	88504
E	06-04-2018	78449
A	04-04-2018	98954
D	04-04-2018	57618
E	04-04-2018	22011
C	02-04-2018	85976
D	02-04-2018	42842
E	02-04-2018	94057
A	05-04-2018	25063
E	05-04-2018	51320
D	06-04-2018	99720
E	03-04-2018	16845

df2

comp	date	adv
C	03-04-2018	980
C	06-04-2018	6183
B	06-04-2018	8077
B	02-04-2018	241
D	02-04-2018	3642
B	04-04-2018	3878
E	02-04-2018	1501
B	03-04-2018	6727
D	06-04-2018	4259
D	03-04-2018	4434
A	06-04-2018	6745
E	03-04-2018	2208

Inner Join

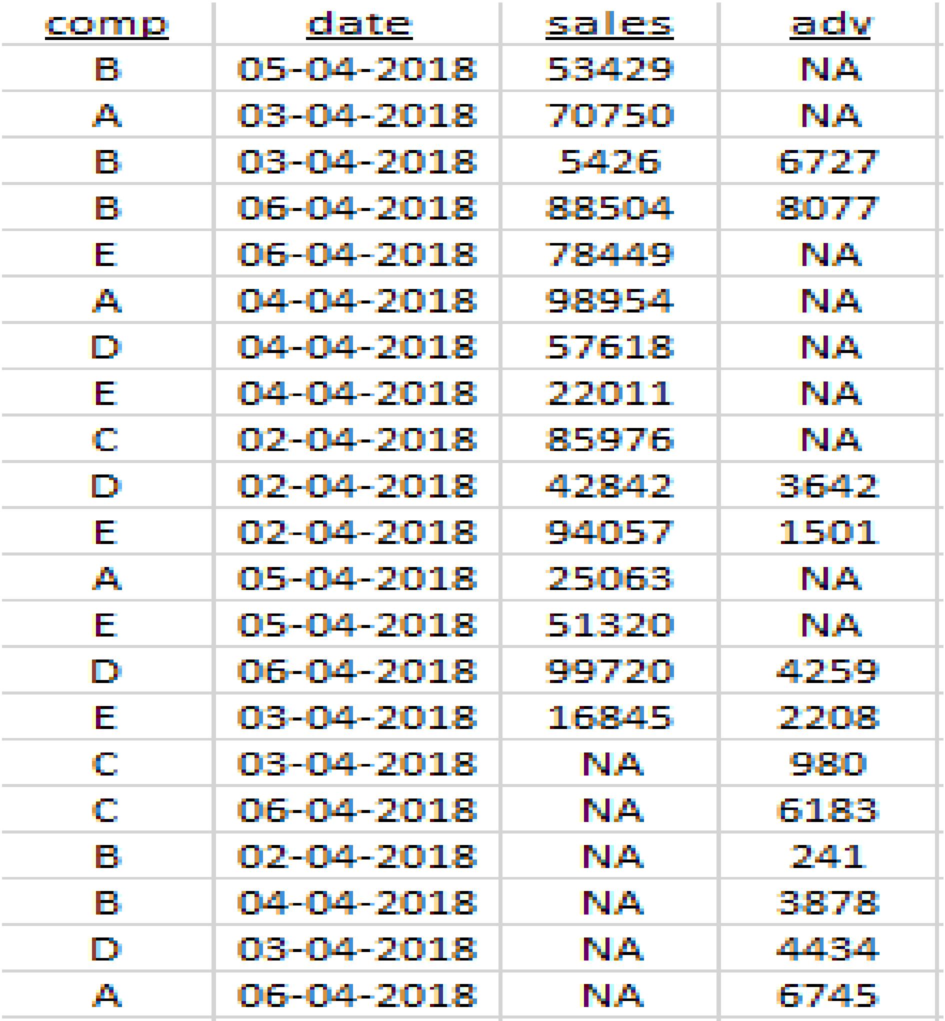
- `dplyr::inner_join(df1, df2);`



comp	date	sales	adv
B	03-04-2018	5426	6727
B	06-04-2018	88504	8077
D	02-04-2018	42842	3642
E	02-04-2018	94057	1501
D	06-04-2018	99720	4259
E	03-04-2018	16845	2208

Full Join

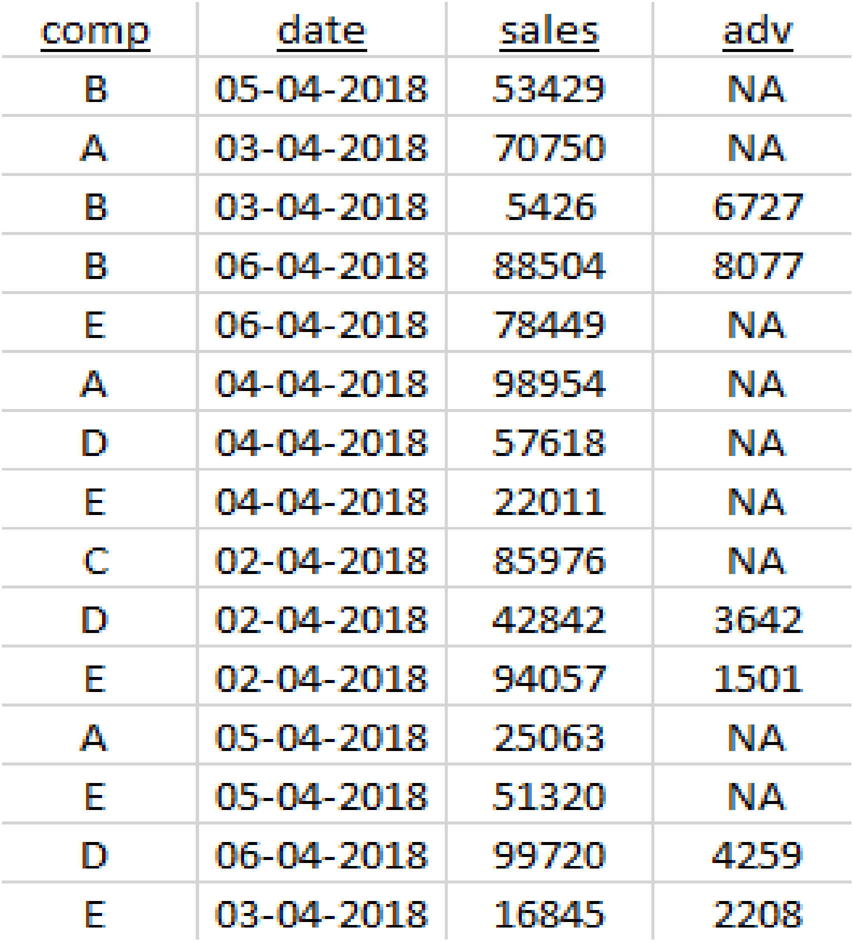
- `dplyr::full_join(df1, df2);`



comp	date	sales	adv
B	05-04-2018	53429	NA
A	03-04-2018	70750	NA
B	03-04-2018	5426	6727
B	06-04-2018	88504	8077
E	06-04-2018	78449	NA
A	04-04-2018	98954	NA
D	04-04-2018	57618	NA
E	04-04-2018	22011	NA
C	02-04-2018	85976	NA
D	02-04-2018	42842	3642
E	02-04-2018	94057	1501
A	05-04-2018	25063	NA
E	05-04-2018	51320	NA
D	06-04-2018	99720	4259
E	03-04-2018	16845	2208
C	03-04-2018	NA	980
C	06-04-2018	NA	6183
B	02-04-2018	NA	241
B	04-04-2018	NA	3878
D	03-04-2018	NA	4434
A	06-04-2018	NA	6745

Left Join

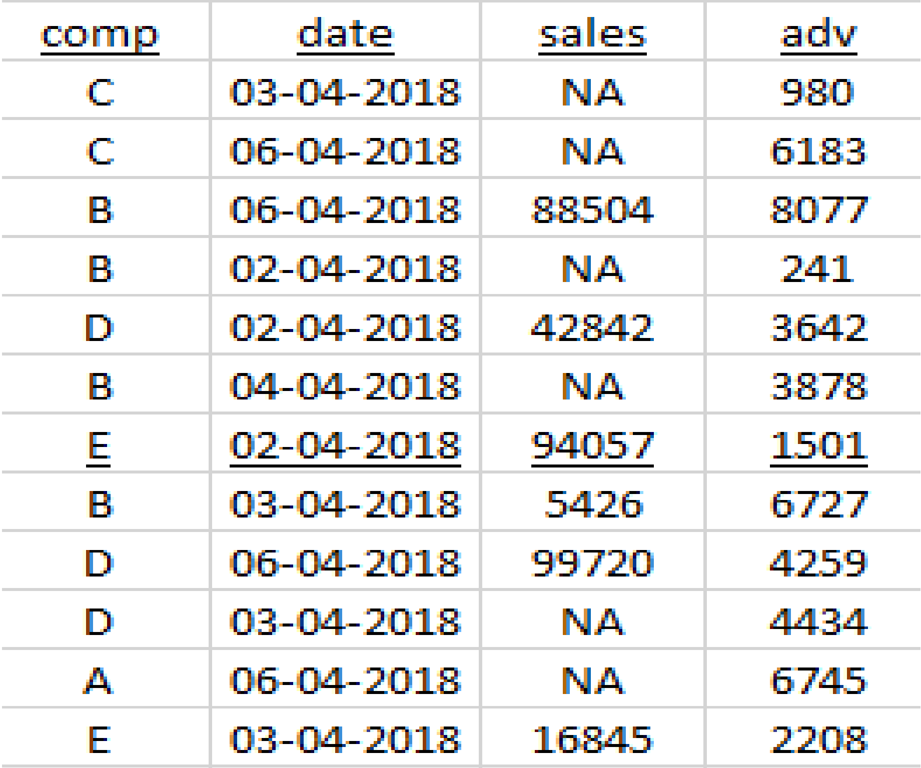
- dplyr::left_join(df1, df2);



comp	date	sales	adv
B	05-04-2018	53429	NA
A	03-04-2018	70750	NA
B	03-04-2018	5426	6727
B	06-04-2018	88504	8077
E	06-04-2018	78449	NA
A	04-04-2018	98954	NA
D	04-04-2018	57618	NA
E	04-04-2018	22011	NA
C	02-04-2018	85976	NA
D	02-04-2018	42842	3642
E	02-04-2018	94057	1501
A	05-04-2018	25063	NA
E	05-04-2018	51320	NA
D	06-04-2018	99720	4259
E	03-04-2018	16845	2208

Right Join

- dplyr::right_join(df1, df2);
  - The above is identical to, dplyr::left_join(df2, df1);



comp	date	sales	adv
C	03-04-2018	NA	980
C	06-04-2018	NA	6183
B	06-04-2018	88504	8077
B	02-04-2018	NA	241
D	02-04-2018	42842	3642
B	04-04-2018	NA	3878
E	02-04-2018	94057	1501
B	03-04-2018	5426	6727
D	06-04-2018	99720	4259
D	03-04-2018	NA	4434
A	06-04-2018	NA	6745
E	03-04-2018	16845	2208

Loop Functions

- Writing loops in a single command. Can come very handy and compact. The function name ends with "apply".
- List of functions: `lapply(), apply(), sapply(), tapply(), mapply()`
- These functions work on a list of inputs, not just one input!
- E.g.
  - `X = list(a = 1:10, b = rnorm(100, 0, 1), c = runif(1e3, 9, 91));`
  - `lappy(X, mean); # returns a list`
  - `sapply(X, mean); # returns a vector`
- Let's say we want to find `cor(X,X^i)` for `i in 1:10` w/o writing a loop?
  - `X = rnorm(1000, 0, 1);`
  - `sapply(1:10, function(i) cor(X,X^i));`
    - Here we have used anonymous function, i.e. a function w/o a name.

- `apply()` is mostly used for applying functions on rows or cols of a matrix
  - Like taking means by rows
  - `M = matrix(1:50, nrow = 10, ncol = 5);`
  - `apply(M, 1, mean); # mean of each row`
  - `apply(M, 2, mean); # mean of each col`
  - You can do the above using a loop also, but `apply()` is more compact.
  - The faster version of above are also available:
    - `rowSums(x)` is equivalent to `apply(x, 1, sum)`
    - `colMeans(x)` is equivalent to `apply(x, 2, mean)`

- `tapply()` is used to apply a function over a subset of vector. Lets say,
  - `x = c(rnorm(100), runif(100));`
  - `f = rep(1:2, each = 100);`
  - `tapply(x, f, mean);`

Pipe and the `dplyr` package

- Download the `tidyverse` package and load it: `library(tidyverse);`
  - Also load `dplyr: library(dplyr);`

- We will use the `gapminder` dataset.
  - Install `gapminder` package
  - `df = gapminder::gapminder; df = as.data.frame(df);`

- Selecting a particular column
  - `df %>% select(lifeExp);`
  - Performing operation on data
    - `df %>% select(lifeExp) %>% mean`

- Selecting some rows (based on a filtering criterion):
  - `df %>% filter(year > 2000)`
  - `df %>% filter(year > 2000 & country == "India")`

- Selecting some rows (based on a filtering criterion):
  - `df %>% filter(year > 2000)`
  - `df %>% filter(year > 2000 & country == "India")`

- Selecting and filtering together
  - `df %>% select(country, year, lifeExp) %>% filter(year > 2000)`
  - `df %>% select(country, year, lifeExp) %>% filter(country == "India")`

- Grouping data
  - `df %>% select(country, year, lifeExp) %>% group_by(lifeExp) %>% as.data.frame %>% head`
  - `df %>% group_by(country, gdpPercap) %>% as.data.frame %>% head`

- Create new columns (mutate)
  - `df %>% mutate(decade = 10*(as.integer(year/10))) %>% head`

- Ordering
  - `df %>% arrange(-pop) %>% head`

- Performing aggregation on grouped data is the most important usage of dplyr library
- Aggregation:
  - `df %>% select(-c(continent)) %>% group_by(country) %>% summarise_all(mean)`
  - `df %>% select(-c(continent)) %>% group_by(country) %>% summarise_all(mean)`
  - `df %>% group_by(continent) %>% summarise_all(mean)`
  - `df %>% group_by(continent, year) %>% summarise_all(mean) %>% filter(continent == "Asia")`
- Counting and ordering in Aggregation:
  - `df %>% group_by(continent, year) %>% summarise(n()) %>% filter(continent == "Asia")`
  - `df %>% group_by(continent) %>% summarise(n())`
  - `df %>% filter(year == 2007) %>% group_by(pop) %>% arrange(-pop)`
  - `df %>% filter(year == 2007) %>% group_by(pop) %>% arrange(-gdpPercap)`
  - `df %>% filter(year == 2007) %>% group_by(pop) %>% mutate(GDP = gdpPercap*pop) %>% arrange(-GDP) %>% as.data.frame %>% select(country, GDP) %>% head(15)`