MIS 572:

Introduction to Big Data Analytics

Data Management and Analysis with R

Yihuang K. Kang

Hello R!

- So, R you ready for R? (remember to check out this article on New York Time).

- Let's start our adventures with R. Enter the following command in RStudio **Console.**

```
print("Hello R!")
[1] "Hello R!"
```

Hello R! (cont.)

- Do some calculations?

```
num <- c(1,2,3,4,5) #use c() to create/combine vectors
sum(num)
[1] 15
mean(num)
[1] 3
prod(num)
[1] 120
num * 10
[1] 10 20 30 40 50
```

- Note that R can directly do the operations on the vector *num*, not just on each value (scalar), as R is a vectorized/array programming language.

Hello R!(cont.)

- R is also a *[functional programming language]*. You can do anything with functions that you can do with vectors—assign them to variables, pass them as arguments to other functions, create functions that return functions, etc.

```
num <- 1:5 # equivalent to “num <- c(1,2,3,4,5)”

# “apply” each function name to
# the anonymous function with num as its input
lapply( c(sum, mean, prod), FUN = function(f) f(num))

[[1]]
[1] 15

[[2]]
[1] 3

[[3]]
[1] 120
```

Course R Package "bigDataR"

- We'll be using a package called "*bigDataR*", which includes all datasets and R function source codes used in this class.

- Our research group keeps updating the package. We will always make it available for students in CM@NSYSU.

```r
# Load all course datasets and functions
library(bigDataR)
```

Operators

Arithmetic	+ Addition	− Subtraction
	* Multiplication	/ Division
	^ or ** Exponentiation	**%%** Modulus
	%/% Integer Division	**:** From:To
	%*% Matrix/Vector Multiplication	**%o%** Outer Product
	%^% Matrix Power/Exponential (see package ***expm***)	
Logical	**!** NOT	**xor()** XOR
	& AND/ vector AND	**&&** Short-circuit/scalar AND
	| OR / vector OR	|| Short-circuit /scalar OR
Relational	>, >= greater then (or equal to)	<, <= less than (or equal to)
	==, != (not) equal to	**%in%** any match in
Other	=, <-, ->, <<- assignment	**::** package::object

Data Type

- R primitive data type

logical	**TRUE** or **FALSE** (**T** or **F**)
integer	**0**, **10** , **-123**, …, etc.
double	**1.234**, **-5.678**,…, etc. (The default data type for numeric values)
complex	**10 + 1i**
character	“**abc**”, “**123**”, “**abc123**”.
Special	**NA, NULL, Inf/-Inf**

Data Type(cont.)

- The ***type*** of data is literally R internal data type of an object. Use *typeof()* to identify.

```
typeof('abc')
[1] "character"
typeof(9i)
[1] "complex"
typeof(123)
[1] "double"
```

- The ***mode*** of data is similar to ***type*** but it is how the data is actually stored in memory (or in your hard drive). Use *mode()* to identify variable "mode" of your dataset.
- Enter *?typeof* and *?mode* for more information.

Try it!

- Let’s say we have a character vector **v**. We’d like to know whether all components/cells are lowercase letters. (hint: use **all()** and built-in constant **letters**)

  E.g.

```
v <- c('a','b','1')
```

- Run the following statements:

```
if(T | (a <- 10) ) print('Hi');
if(T || (a <- 20) ) print('Hi');
```

Is a == 20 TRUE? If not, why not?

Use "<-" or "=" as R assignment operator?

- You may notice that R has more assignment operators than other computer languages (e.g. as C and Java). R uses "<-" as its default assignment for some historical reasons (recall that R is from S language). Many R books and experts (especially those with Statistics background), recommend using "<-". Computer scientists, however, prefer "=", most likely because "=" is commonly used in modern languages (or simply because it's one less keystroke!).

- Note that "=" is also used in function parameter passing. For example,

```
mean(x = 1:10) # "=" is used as a function argument binding
mean(x <- 1:10) # "<-" is used as a variable assignment
```

  So, the answer to this question is "It's up to you". Just don't confuse yourself. For those of you who are familiar with modern scientific programming languages, such as Python and MATLAB, you may stick to "=". If you are new to this field and have no preference, just use "<-" for compatibility sake.

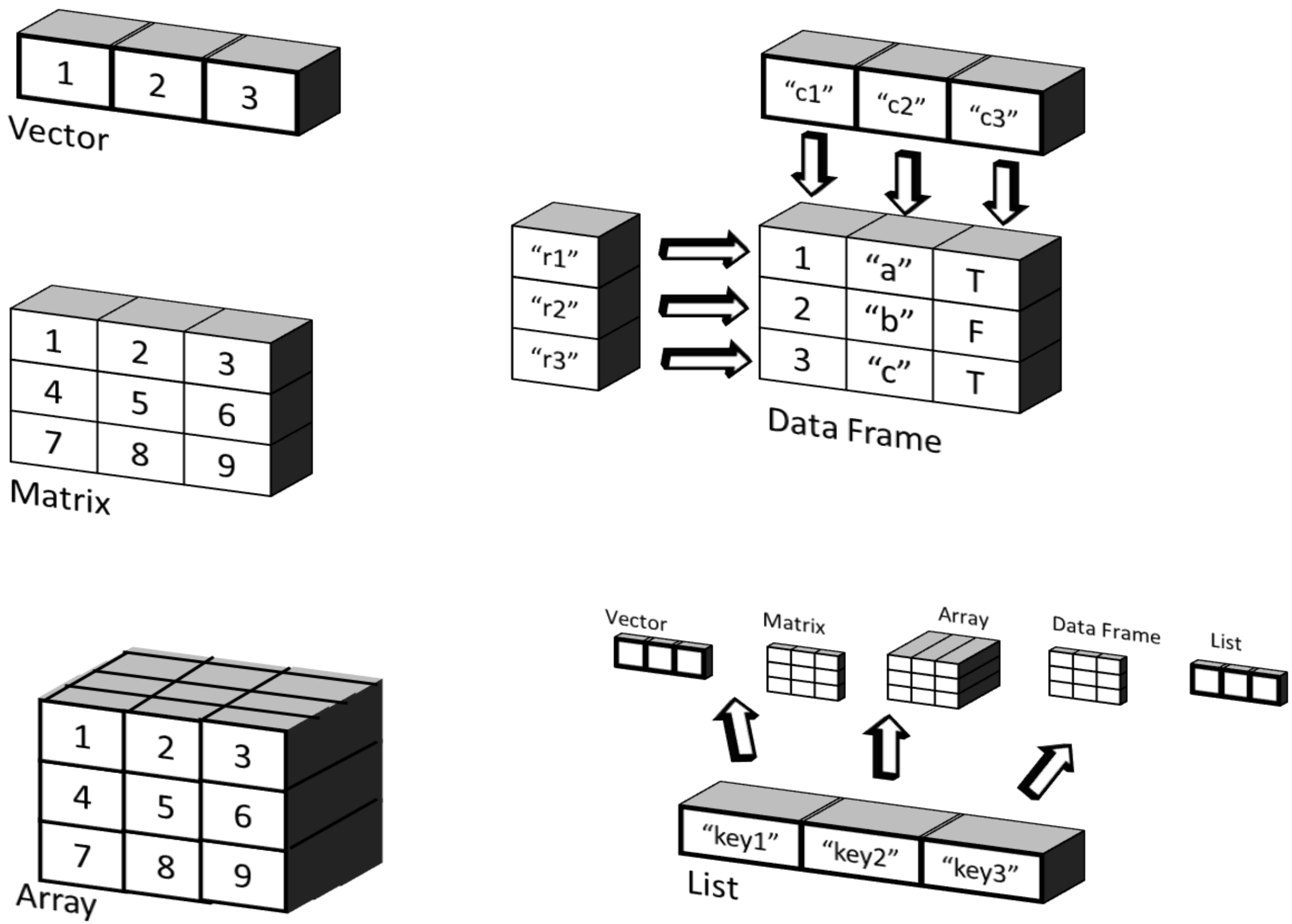
A bit about "Levels of Measurement"

- Levels (Scale) of measurement here is classification of information or value of a variable:
  - ***Nominal*** variable has values that are distinct symbols/labels. No relation, such as ordering or distance, is implied. E.g. Gender (Male, Female).
  - ***Ordinal*** variable is similar to nominal variable but we can rank order the labels. Note that there is still no "distance" notion among labels. E.g. Temperature (Cool, Warm, Hot).
  - ***Interval*** variable has values that the order and distance among them make sense. But the distances are fixed and any mathematical operations are not allowed. For example, we normally don't say temperature 10 degree (in C) is "twice colder" than 20 degree.
  - ***Ratio*** variable allows real numbers that makes any operations, such as ratios and differences, logical. E.g. cost of living in USD.

A bit about "Levels of Measurement"(cont.)

- Later in this class, however, we consider more general terms of the measurement—discrete (categorical) and continuous (numeric) variables.
  - **Discrete variables** include the aforementioned nominal and ordinal variables. In R, we can declare a variable as a *factor*, a special data type that defines ordered or unordered *levels* with *labels* (possible values of a variable). We'll soon discuss more about it.
  - **Continuous variables** have real and numeric values that can be used in any aforementioned mathematical operations. In R, numeric vectors are treated as continuous.

Data Structure



Data Structure(cont.)

- **Vectors** are 1-dimentional containers that can store <u>the same mode </u>of data.

```
v1 = 1:10
v1
[1] 1 2 3 4 5 6 7 8 9 10
v2 = c("a","b","c","1","2","3")
v2
[1] "a" "b" "c" "1" "2" "3"
v3 = vector(mode="character", length= 3)
v3
[1] "" "" ""
v4 = vector(mode="logical", length= 2)
v4
[1] FALSE FALSE
```

Data Structure(cont.)

- **Matrices** are similar to Vectors but have <u>two</u> dimensions. A special property of a Matrix is that the names of columns & rows can be assigned.

```
m = matrix(1:4, ncol=2, nrow=2, byrow = T)
m
     [,1] [,2]
[1,]    1    2
[2,]    3    4
colnames(m) = c('c1','c2'); rownames(m) = c('r1','r2');
m
   c1 c2
r1  1  2
r2  3  4
t(m) # transpose the matrix
   r1 r2
c1  1  3
c2  2  4
```

Data Structure(cont.)

- **Arrays** can be considered generalizations of Vectors and Matrices with arbitrary numbers of dimensions.

```
a = array(c(1,2,3,4,5,6), dim=c(1,2,3)) # 1 by 2 by 3
a
, , 1
     [,1] [,2]
[1,]  1    2
, , 2
     [,1] [,2]
[1,]  3    4
, , 3
     [,1] [,2]
[1,]  5    6
dim(a) # get the dimension of the array
[1] 1 2 3
```

Data Structure(cont.)

- **Data Frame** may be the most frequent data type you will use in R. Similar to the Matrices, they're 2-dimentional but columns/variables can be in different *modes*.

```
 v1 = c(9,8,7)
 v2 = c("a","b", "c")

 #combine 2 vectors
 d = data.frame(x1 = v1, x2 = v2, stringsAsFactors = F)
 str(d) # show the structure of the data frame "d"

'data.frame':  3 obs. of  2 variables:
$ x1: num  9 8 7
$ x2: chr  "a" "b" "c"
```

Data Structure(cont.)

- Another data type we could use when dealing with massive datasets is **data.table**, one of the most popular R packages on GitHub. You may consider replacing data frame with the data.table. Note that, however, some R packages & functions do not support data.table. We will discuss more about it later in this class.

```
# Using package "data.table"
library(data.table)
d_dt = data.table(d)

str(d_dt) # "data.table" is also a kind of data frame
Classes ‘data.table’ and 'data.frame':  3 obs. of  2
variables: $ x1: num  9 8 7 $ x2: chr  "a" "b" "c"

d_dt[, c("v1") :=  v1 * 10, ]
```

Data Structure(cont.)

- **Lists** are special key-value pairs of data. Such pair in Lists can also be considered a column of a data frame with the key as the column name.

```
L = list(k1 = c(9, 8, 7),
         k2 = c("a","b","c"), k3 = c(1))
L
$k1
[1] 9 8 7
$k2
[1] "a" "b" "c"
$k3
[1] 1
L$k1  # get the data in the list L with the key “k1”
[1] 9 8 7
L$k3
[1] 1
```

Data Structure(cont.)

- **Factors** are practically used to represent the ordinal and nominal variables. They are literally the term *factor* with *levels* commonly found in Statistics worlds.
- The most intuitive way is to consider Factors as sequential integer vectors with strings as formats/codebook indicating what these numbers actually mean.

```
group = c("control", "treatment", "treatment", "control")
group
[1] "control" "treatment" "treatment" "control"
group_f = factor(group, levels=c("treatment", "control"))
group_f
[1] control treatment treatment control
Levels: treatment control
typeof(group_f)  # check the actual data type of “group_f”
[1] "integer"
unclass(group_f) # remove class attributes to get real data values
[1] 2 1 1 2
attr(,"levels")
[1] "treatment" "control"
```

Data Structure(cont.)

- **Tables** are R's implementation of frequency tables, also called *contingency tables or crosstabs* for 2-D table, commonly used when analyzing categorical data. Tables are often created by *table*() and *xtabs*().

```
library('bigDataR')
smoker = as.data.frame(smoker)
tbl_smoker = table(smoker$Smoke, smoker$SES)
table(smoker$Smoke, smoker$SES)
           High Low Middle
  current    51  43     22
  former     92  28     21
  never      68  22      9

chisq.test(tbl_smoker)# chi-square test of independence
      Pearson's Chi-squared test

data:  tbl_smoker
X-squared = 18.5097, df = 4, p-value = 0.0009808
```

Data Structure(cont.)

- Two frequent data type operations are type *verification* and *casting*. R provides a set of functions that help you verify and convert all the aforementioned data types.
- Use is.*() functions for type verification. E.g. is.vector(), is.array(), is.data.frame(), ..., etc.
- Use as.*() functions for type casting. E.g. as.vector(), as.array(), as.data.frame(), ..., etc.

```
a = c(1,2,3,4) ; a
[1] 1 2 3 4
is.vector(a)
[1] TRUE
is.matrix(a)
[1] FALSE
m = as.matrix(a)
is.matrix(m)
[1] TRUE
t(m) # transpose the matrix
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

Data Structure(cont.)

- Missing values–**NA** vs. **NULL**
  - NA is a special logical data type that indicates whether a value (in a cell of any aforementioned container) is existent but missing. NULL, however, indicates the value is nonexistent and its data type is unknown (null).
  - NA is usually used to represent missing values, whereas NULL is used in data management and function arguments.

```
a = NA ; typeof(a)
[1] "logical"
object.size(a) # get the object size (bytes) in memory
48 bytes
b = NULL; typeof(b)
[1] "NULL"
object.size(b)
0 bytes
```

Function

- Again, everything in R is an *object*! An R function is actually a special data type (class) called *function*.
- R function can take multiple named inputs and return an object (with *return()*) in any data type.

```
square = function(x = NULL){
 return(ifelse(is.null(x), "NULL", x^2 ) );
} # if x is not NULL, then output x squared.
square() # if there’s no input (x is NULL)
[1] "NULL"
square(3) # if x = 3
[1] 9
# explicitly define value for named input  “x”
square(x = 10)
[1] 100
```

Function(cont.)

- Variable Scope in R
  - Variables (objects) created by body of function statements are only visible within the function. They are described as local objects. On the other hand, variables created outside functions are called global objects and are accessible anywhere in the same R session/instance of the operation system.

```
x = 'global_x'
printXY = function(){
 y = 'local_x'; print(x); print(y);
}
printXY()
[1] "global_x"
[1] "local_x"
x
[1] "global_x"
y
Error: object 'y' not found
```

Function(cont.)

- Function dot ellipsis (unknown # of inputs)
  - In the case that we would like a function that takes unknown numbers of inputs/parameters, R dot ellipsis (reserved words **...** and **..1**, **..2**, **...n**) could help. Here, " **...** " stands for all input values , whereas "**..n**" extracts ***n***th argument value.

```
# Set operations, an R ellipsis example
setOper = function(f, ...){
  el = list(...)
  return(Reduce(f, el))
}
setOper(intersect, 1:5, 2:6, 3:5)
[1] 3 4 5
setOper(union, 1:5, 2:6, 3:5)
[1] 1 2 3 4 5 6
```

Control Statement

- **if-else**

```
f = function(x){
   if(x > 10){
     print("x is great than 10");
   } else if( x >=0 & x <=10) {
     print("x is between 0 and 10");
   } else {
     print("x is less than 0");
   }
}
f(-1)
[1] "x is less than 0"
f(5)
[1] "x is between 0 and 10"
f(11)
[1] "x is great than 10"
```

Control Statement(cont.)

- ***ifelse()***
  - DO NOT use if-else statements for simple assignment purposes. Use *ifelse*() instead or simply vectorize your assignments!

```
library(ggplot2movies)
movies = as.data.frame(movies); movies$longshort = ""
# Very bad practice with for loop. Don't do this!
system.time({
  for(i in 1:nrow(movies)){
     if(movies[i, "length"] > 120)movies[i, "longshort"] = "long"
     else movies[i, "longshort"] = "short" }
})
# Use ifelse() instead
system.time(
   movies$longshort <- ifelse(movies$length > 120, "long", "short"))

# Or simply vectorized it!
system.time({
  movies[movies$length > 120, "longshort"] = "long"
  movies[movies$length <= 120, "longshort"] = "short"
})
```

Control Statement(cont.)

- **for** loop

```
for(i in c('a','b')) print(i)
[1] "a"
[1] "b"

for(k in 1:3) {
  if(k == 3) break; print(k);
}
[1] 1
[1] 2
```

Always avoid using loop for unnecessary iterations, as using nested loops or loop with thousands(+1000, for example) of iterations are extremely inefficient in R. Such looping practices do not convey any higher level goal of data analysis. Try to vectorize or "functionalize" your code!

Control Statement(cont.)

- **while** and **repeat (do)** loops

```
x = 3;
while(x > 0){
 print(x);
 x = x - 1;
}
[1] 3
[1] 2
[1] 1
x = 3
repeat{
 print(x);
 x = x - 1;
 if(x < 1) break;
}
[1] 3
[1] 2
[1] 1
```

Control Statement(cont.)

- ***switch*(*expr*, …)**
  - R provides *switch*() function instead of switch/select statements found in most of computer languages. The *switch*() also works differently.
  - For numeric *expr*, *switch*() evaluates *expr* as the numeric index to following statements separated by "," .
  - For character *expr*, however, *switch*() evaluates *expr* as the character key.

Control Statement(cont.)

- ***switch(expr, …)***

```
switch(3, 'a' = {x = x + 5;},
          'b' = {x = 999},'c' = {x = 'ABC'})
 x # 3nd statement after expr
 [1] "ABC"
 x = 0
 switch(1, 'a' = {x = x + 5;},
       'b' = {x = 999}, 'c' = {x = 'ABC'})
 x # 1st statement after expr
 [1] 5
 switch('a', 'a' = {x = x + 5;},
       'b' = {x = 999}, 'c' = {x = 'ABC'})
 x # statement with key value 'a'
 [1] 10
```

Check *recode*() in **car** package for a more "select case"-like function.

Try It!

- ❑ Create your own R function that discretizes numeric BMI numbers into following categories/levels:

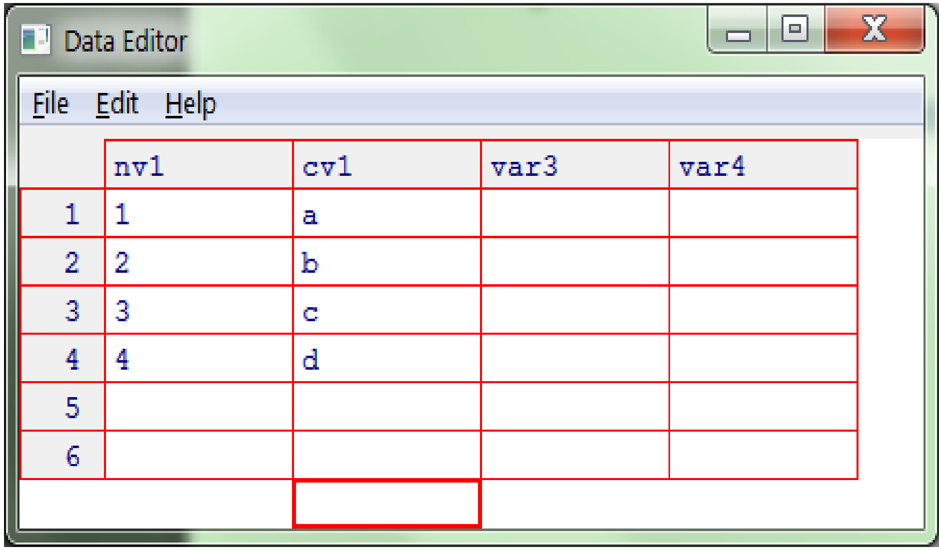
  Underweight: **<18.5**
  Normal weight: **18.5 to <25**
  Overweight: **25 to <30**
  Obesity: **30 or greater**

- ❑ Use *xtabs*() or *table*() to generate a frequency table of the BMI categories. You can also use *cut*() or *car::recode*() instead, if you find them easier for you.

Data Input and Output

- Let enter some data in R. Most intuitive way to do it in R is to use default data editor. Enter *edit()* in RStudio Console if you use RStudio Desktop on MS Windows.

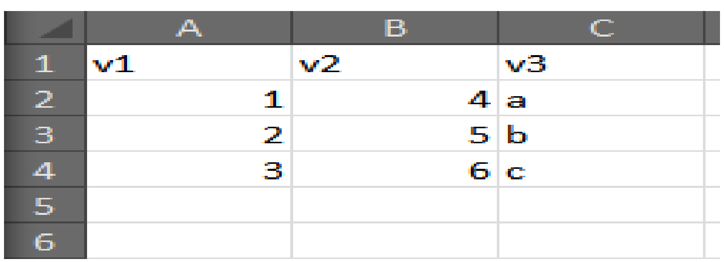
```
mydata = data.frame()
mydata = edit(mydata)
mydata
   nv1 cv1
1   1    a
2   2    b
3   3    c
4   4    d
```

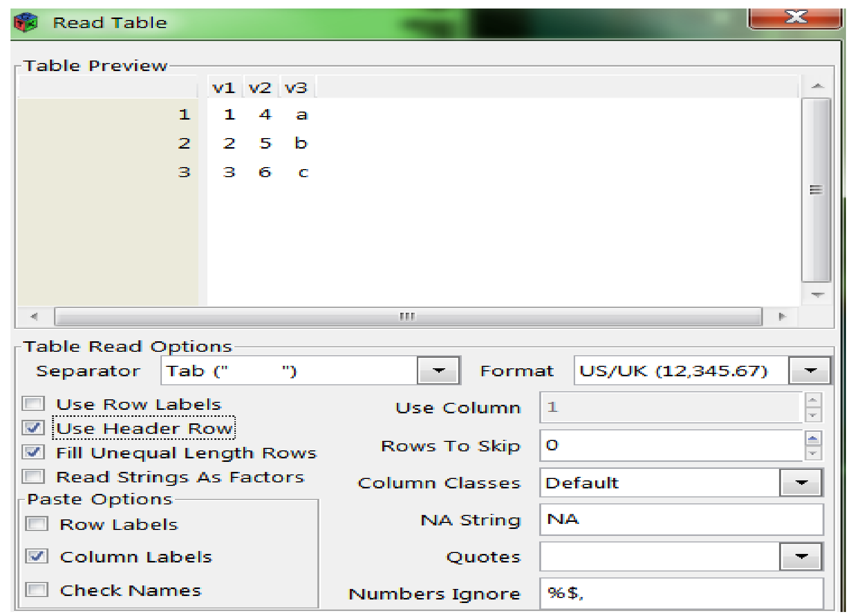


Data Input and Output(cont.)

- Another advanced data editor, *RGtk2Extras::dfedit()*, allows you to directly modify your data frames, alter variable/column data types, and copy & paste your data from other editors (e.g. notepad) or spreadsheets.

```
library("RGtk2Extras")
df_data = data.frame()
dfedit(df_data, modal=F)
```





- Also check out the dataset editor in *Rcmdr* (R Commander). Note that, again, these data editors only work on RStudio Desktop.

Data Input and Output(cont.)

- R provides *read.*()* functions that help import your data in plain text (e.g. CSV). Let's say we have an CSV file called *vars.csv* as below.

```
'var1', 'var2'
101, 'a'
102, 'b'
103, 'c'
```

```
vars_data = read.csv(file='vars.csv', header=T, quote="'")
vars_data
  var1 var2
1  101   a
2  102   b
3  103   c
```

- Check out package *foreign* if you're interested in how R read datasets created by other scientific computing software. For those of you who are SAS users, see package *sas7bdat.parso* if you're dealing with large SAS files.

Data Input and Output(cont.)

- Function *write.*()* are used to output your data.

```
library("foreign")
write.foreign(df = vars_data, datafile='vars_data',
 codefile="spss_read_code.txt" , package=c("SPSS"))
file.show('vars_data') # Export as SPSS files
file.show('spss_read_code.txt')
```

- You can also save objects in current working directory as R's own format (image file with extension *.RData*) by using *save()*.

```
save.image(); # Save everything in current Environment
save(x=vars_data, file='vars_data.RData'); # Save “vars_data”
rm(list=ls()) #Remove all objects
load("vars_data.Rdata") #load the image file we saved
```

- Enter *?save* and *?load* for more information.

Data Input and Output(cont.)

- R can also work with different structured data sources (e.g. RDBMS) by using built-in package *DBI* and other 3rd party database driver packages. Consider working with MySQL (say we have pre-installed package *RMySQL*) as the example.

```
library("DBI")
library("RMySQL")
# create connection
mycon = dbConnect(dbDriver('MySQL'), dbname='testdb',
        host='127.0.0.1', user='root');
# using R SQL pass-through facility:
dbSendQuery(mycon, "create table test (var1 int, var2 char(8))");
dbSendQuery(mycon, "insert into test values(101,'a') ");
test_data = dbGetQuery(mycon, "select * from test");
test_data
   var1  var2
   101   a
dbDisconnect(mycon); # close connection
```

Think "Big"!

- For efficiency's sake, always perform data pre-processing (in RDBMS, NoSQL, or other types of databases) before load huge dataset into local R environment.

- Any big dataset that cannot fit in memory might not work in R. (e.g. ~ 2GB for 32-bit OS)

- Check out package *data.table*, *sqldf*, *parallel*, *ff*, *bigmemory*, *doMC*, *snow*, *Rhadoop*, *and* *h2o*. We'll soon discuss how R deal with big datasets.

Think "Big"! (cont.)

- Two more efficient functions to import plain text data file: *data.table::fread()*, which takes advantage of big memory. *sqldf::read.csv.sql()*, which uses facilities of RDBMS (SQLite by default) to read the file and speed up the import process.

```
write.csv(as.data.frame(matrix(runif(10 ^ 6 ,0,1),
        nrow=1000)), file='rnum.csv'); # An ~18 MB CSV file.
file.info("rnum.csv")$size; # get file size
system.time({rnum = read.csv(file= "rnum.csv", header=T)});

rm(rnum); library("data.table"); #load package data.table
system.time({ rnum = fread(input="rnum.csv")});

rm(rnum); library("sqldf"); # load package sqldf
system.time({ rnum = read.csv.sql(file="rnum.csv",
            header=T,sep = ",");});
# The elapsed times vary on different machines
```

Data Manipulation

- Filtering

```
mtcars = data.frame(mtcars)
mtcars[grep('Toyota', rownames(mtcars)),] # show me “Toyota”
                 mpg cyl disp hp drat wt qsec vs am gear carb
Toyota Corolla 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1
Toyota Corona  21.5 4 120.1 97 3.70 2.465 20.01 1 0 3 1

mtcars[,c('mpg','hp','wt')]; # select columns
          mpg hp wt
Mazda RX4 21.0 110 2.620
Mazda RX4 Wag 21.0 110 2.875
Datsun 710 22.8 93 2.320
…
mtcars[mtcars$wt > 5, c('mpg','wt')] #please also check subset()
                   mpg wt
Cadillac Fleetwood 10.4 5.250
Lincoln Continental 10.4 5.424
Chrysler Imperial 14.7 5.345
```

Data Manipulation(cont.)

- Sorting and ranking

```
set.seed(1) # set random seed
rand = runif(n=5,min=0,max=1)
rand # generated random numbers
[1] 0.265509 0.372124 0.572853 0.908208 0.201682
sort(rand)
[1] 0.201682 0.265509 0.372124 0.572853 0.908208
```

```
# sort by descending “wt”
mtcars[order(mtcars$wt, decreasing=T),c('mpg','wt')]
# sort by ascending “cyl” and descending “wt”
mtcars[order(mtcars$cyl, -mtcars$wt),]
# rank by "mpg“ & create a new dataset
data.frame("car_name" = rownames(mtcars),
    "mpg" = mtcars$mpg,
    "rank" = rank(mtcars$mpg,ties.method = "first"));
```

Data Manipulation(cont.)

- Removing duplicate records

```
 dup = data.frame(x1=c('a','a','b','b','c'), x2=c(1,1,2,3,4))
 dup
  x1 x2
1  a  1
2  a  1
3  b  2
4  b  3
5  c  4

# Remove duplicates and keep all columns
dup_removed = dup[!duplicated(dup), ]
dup_removed
  x1 x2
1  a  1
3  b  2
4  b  3
5  c  4
```

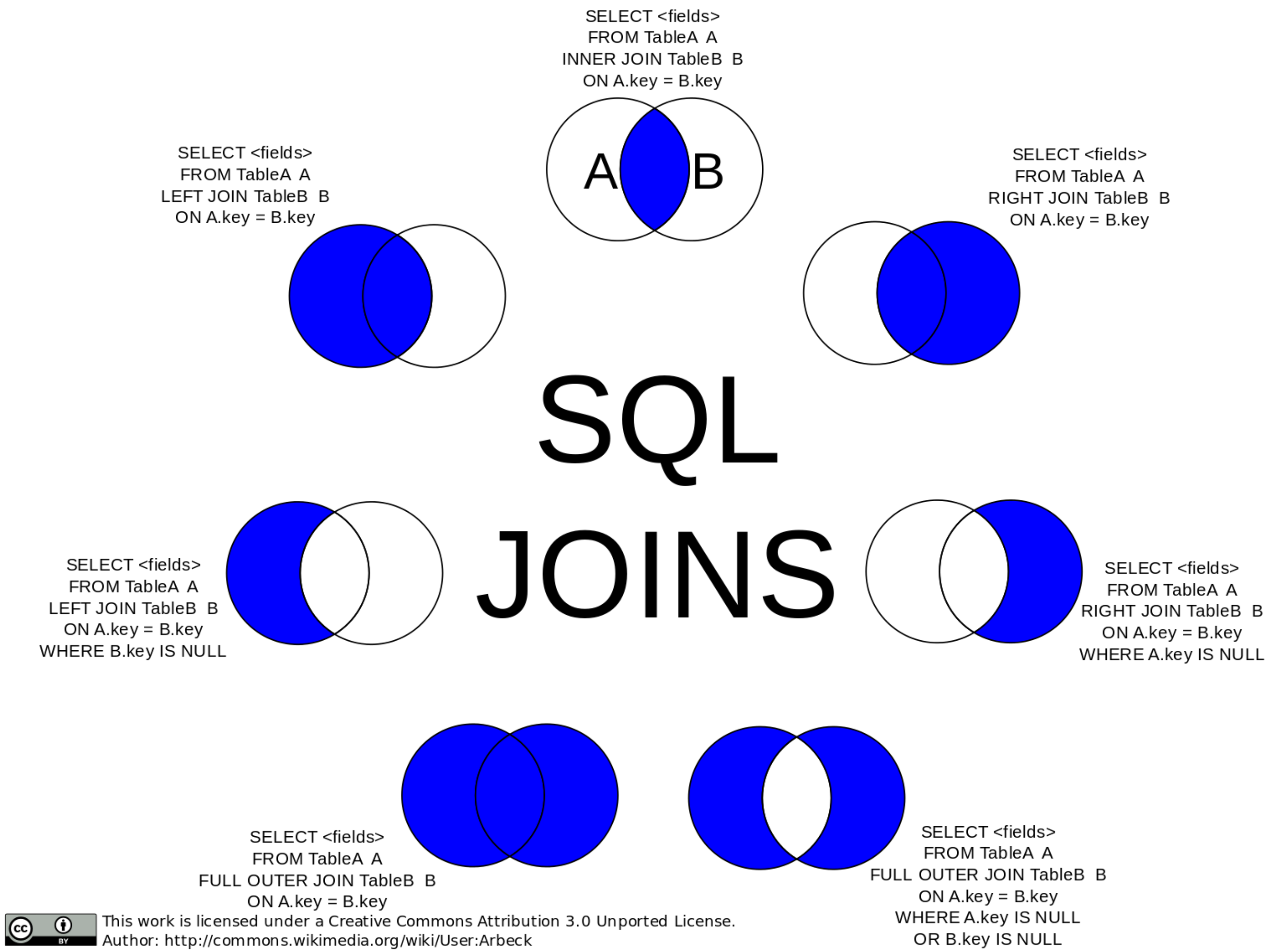
Data Manipulation(cont.)

- Deleting and adding columns/variables

```
mtcars[,c('vs','am','gear','carb')] = list()
mtcars #some variables/columns have been removed
mtcars = mtcars[, ! colnames(mtcars)
          %in% c('cyl','disp','drat','qsec')]
mtcars
# Or, we can use "NULL“
mtcars$mpg = NULL;
mtcars

# Add a new column
# data frame is a sort of “List”
mtcars$newVar = c(1:32)
mtcars # a new column “newVar” has been added
```

Data Manipulation(cont.)



Data Manipulation(cont.)

- Merging I – Inner Join

```
A = data.frame(id=c(1,3,5), A_val=c('a','x','c'))
A
  id A_val
1  1   a
2  3   x
3  5   c
B = data.frame(id=c(3,5,6), B_val=c('x','y','z'))
B
  id B_val
1  3   x
2  5   y
3  6   z
# inner join by “id”
A_B = merge(x=A, y=B, by.x='id', by.y='id') A_B
  id A_val B_val
1  3   x     x
2  5   c     y
```

Data Manipulation(cont.)

- Merging II – Outer Join

```
 left_A_B = merge(x=A, y=B, by.x='id', by.y='id', all.x=T)
 left_A_B # A left join B
  id  A_val B_val
1  1  a     <NA>
2  3  x      x
3  5  c      y
 right_A_B = merge(x=A, y=B, by.x='id', by.y='id', all.y=T)
 right_A_B # A right join B
  id  A_val B_val
1  3  x     x
2  5  c     y
3  6  <NA>  z
 full_A_B = merge(x=A, y=B, by.x='id', by.y='id', all=T)
 full_A_B  # A full outer join B
  id  A_val B_val
1  1  a     <NA>
2  3  x     x
3  5  c     y
4  6  <NA>  z
```

Data Manipulation(cont.)

- Merging III – Concatenation

```
 colnames(A) = colnames(B) = c('id','val');
 rbind(A,B) # Concatenate vertically
id val
1 1 a
2 3 x
3 5 c
4 3 x
5 5 y
6 6 z
 cbind(A,B) # Concatenate horizontally
  id val id val
1  1   a  3   x
2  3   x  5   y
3  5   c  6   z
```

Data Manipulation(cont.)

- Merging IV – Set Operation

```
# A intersect B
 subset(A, (A$id %in% B$id & A$val %in% B$val))
id val
2 3 x
# A except B
 subset(A, ! (A$id %in% B$id & A$val %in% B$val))
id val
1 1 a
3 5 c
# B except A
 subset(B, ! (B$id %in% A$id & B$val %in% A$val))
id val
2 5 y
3 6 z
```

You may notice that it is ugly and inefficient (although a bit flexible) to do observation-level set operations with base R. Good news is that there's always some R packages that do better jobs!

Data Manipulation(cont.)

- Merging V – Set Operation with package *dplyr*

```
# More efficient row/column binding
dplyr::bind_rows(A, B); dplyr::bind_cols(A, B)

# A intersect/union B
dplyr::intersect(A, B); dplyr::union(A, B);

# A except B; B except A
dplyr::setdiff(A, B); dplyr::setdiff(B, A)

# Observation-level set comparison
dplyr::setequal(A, B)
FALSE: Rows in x but not y: 3, 1. Rows in y but not x: 3, 2.
```

Remember to check out super-fast observation-level set operation functions in *data.table* 1.9.7+ by typing *?data.table::setops*!

Data Manipulation(cont.)

• Using SQL I

The package *sqldf* provide R users advanced data manipulation functions by writing standard SQL codes. As *sqldf* uses SQLite as the backend database by default, any SQL syntax that SQLite supports can be used in *sqldf*().

```
library(sqldf)
# Inner join
sqldf('select A.id, A.val as A_val, B.val as B_val
        from A inner join B on A.id = B.id')
# Left outer join
sqldf('select A.id, A.val as A_val, B.val as B_val from A left
       join B on A.id = B.id')
# Union
sqldf('select * from A union select * from B')
```

Check out [here]() for more information about *sqldf*.

Data Manipulation(cont.)

- Using SQL II

```
 sqldf('select * from A except select * from B')
   id val
1  1   a
2  5   c
 sqldf('select * from A intersect select * from B')
   id val
1  3   x
 sqldf('select * from
       (select * from A union all select * from B) where id > 5')
   id val
1  6   z
 mtcars = data.frame(mtcars);
 sqldf('select row_names, mpg, cyl, wt from mtcars
        where row_names like "%Toyota%" ', row.names=T)
              mpg   cyl   wt
Toyota Corolla 33.9  4     1.835
Toyota Corona  21.5  4     2.465
```

Data Reshaping

- R package *reshape2* by [Wickham]provides a set of functions for you to convert datasets from "wide" to "long" format or "long" to "wide" format using *melt*()/*cast*() functions.

ID	time	var1	var2
1	t1	2	3
1	t2	4	1
1	t3	2	3
2	t1	1	2
2	t3	3	4
3	t1	2	3

melt() →

← ***cast()**

ID	time	variable	value
1	t1	var1	2
1	t2	var1	4
1	t3	var1	2
2	t1	var1	1
2	t3	var1	3
3	t1	var1	2
1	t1	var2	3
1	t2	var2	1
1	t3	var2	3
2	t1	var2	2
2	t3	var2	4
3	t1	var2	3

Data Reshaping(cont.)

- The *melt*() function is used to restructure your dataset into a long format, where each selected variable is in its own row, along with one or a set of ID variables required to uniquely identify the row. The ID variables are usually IDs, time points, or locations of your observations.

```
library("bigDataR"); data("reshape_data");
md = melt(data = reshape_data, id.vars = c("ID", "time")); md
    ID time variable value
1   1   t1     var1     2
2   1   t2     var1     4
3   1   t3     var1     2
4   2   t1     var1     1
5   2   t3     var1     3
6   3   t1     var1     2
7   1   t1     var2     3
8   1   t2     var2     1
9   1   t3     var2     3
10  2   t1     var2     2
11  2   t3     var2     4
12  3   t1     var2     3
```

Data Reshaping(cont.)

- The **cast*() functions, on the other hand, are used to transpose your "melted" dataset into a wide format by following R *formula* definition:

$$rowVar1 + rowVar2 + \ \ldots + rowVarN \ \sim colVar1 + colVar2 + \ \ldots + coVarN$$

where *rowVars* define the crossed variables that uniquely identify each row, and the level combinations of *colVars* create the new columns/variables.

```
# Restore the melted dataset
dcast(data = md, formula = ID + time ~ variable,
   value.var = "value");
```

- We'll soon discuss the R *formula* data type.

Data Reshaping(cont.)

- One advanced feature of **cast*() is that you can specify measure/variable with an aggregation function.

```
# "ID" by "time + variable". Default aggregation function
# length() is used to count the number of rows
dcast(data = md, formula = ID ~ time + variable,
      value.var = "value", fun.aggregate = length)
  ID t1_var1 t1_var2 t2_var1 t2_var2 t3_var1 t3_var2
1  1       1       1       1       1       1       1
2  2       1       1       0       0       1       1
3  3       1       1       0       0       0       0

# "ID" by "variable" with aggregation function mean()
dcast(data = md, formula = ID  ~ variable,
      value.var = "value", fun.aggregate = mean)
  ID   var1    var2
1  1 2.66667 2.33333
2  2 2.00000 3.00000
3  3 2.00000 3.00000
```

Try It!

- Load built-in dataset *CO2*, create two new variables *nonchilled* and *chilled*, count the number of occurrences for both treatments of different plants.

Plant	nonchilled	chilled

- What are the average *uptakes* for different treatments and the origin of the plants?

Type	nonchilled	chilled

"Divide each difficulty into as many parts as is feasible and necessary to resolve it."

— [René Descartes]

Data Aggregation

- One of daily data management tasks is to get summarized group statistics—**data aggregation**. R provides various data aggregation functions that help you shape your data. Take function *aggregate()* and *by()* as the examples.

```
 data(mtcars)
 aggregate(x=mtcars$mpg, by=list("cyl"= mtcars$cyl), FUN=mean)
   cyl  x
1  4    26.66364
2  6    19.74286
3  8    15.10000
 by(data=mtcars$mpg, INDICES=list("cyl"= mtcars$cyl), FUN=mean)
cyl: 4
[1] 26.66364
-----------------------
cyl: 6
[1] 19.74286
-------------
cyl: 8
[1] 15.1
 # equivalent to SQL 'select cyl, avg(mpg) from mtcars group by cyl'
```

Data Aggregation(cont.)

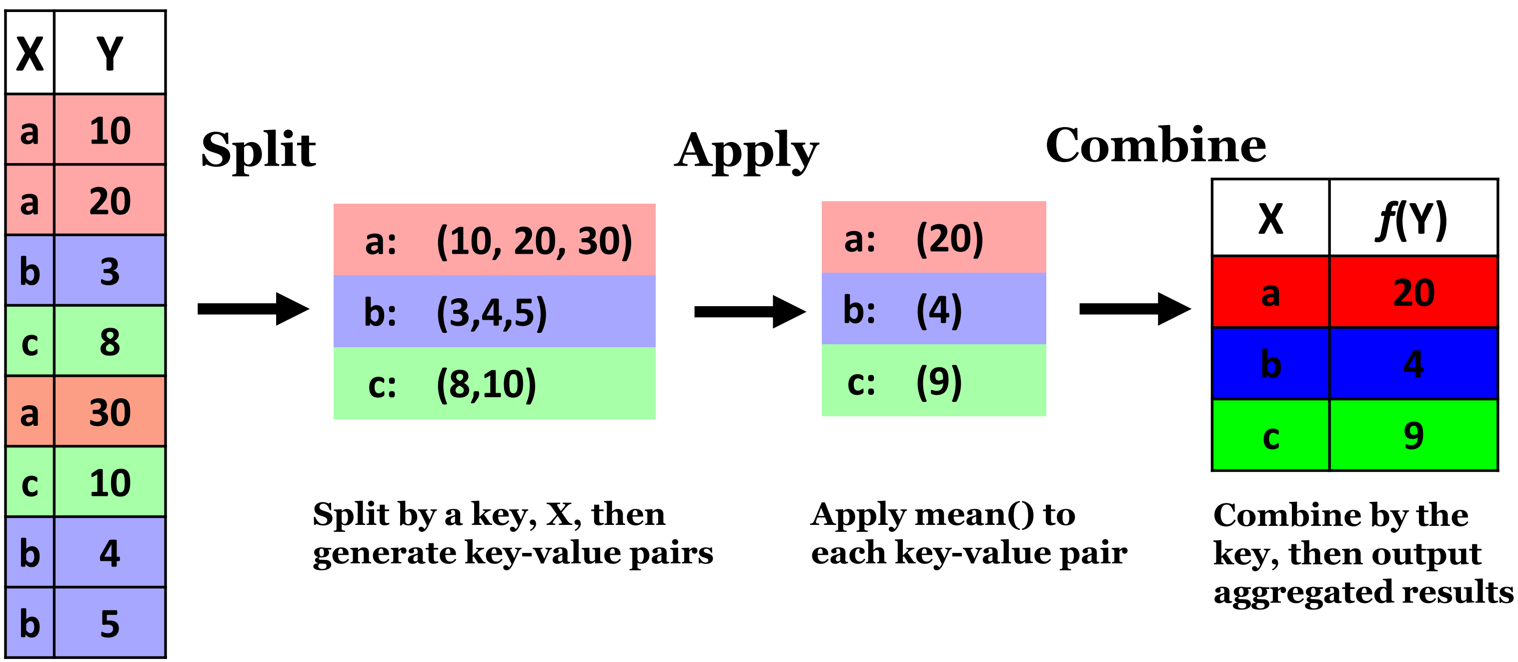
- Not all functions, including your own functions, in R support vectorized computations. You can write a lot of nested loops as you might do with other computer languages. In R, however, you don't have to do this in such inefficient way.
- The *apply()* family functions (*apply*, *lapply*, *sapply*, etc ...) in R provide you a concise and intuitive way to achieve the vectorization instead of using loops.

```
m = matrix(1:16,ncol=4)
apply(m, MARGIN=1,FUN=sum) # get sums by row
apply(m, MARGIN=2,FUN=sum) # get sums by column

set.seed(1); # set seed
L=list( num = 1:10 , randNum = runif(100,0,1) )
lapply(X=L, FUN=mean) # get means for "keys" in the list
$num
[1] 5.5
$randNum
[1] 0.517847
```

Data Aggregation(cont.)

- You may notice that SQL could cope with the summary tasks with elegant syntax. Why bother using R aggregation functions? One major reason is that you can create your own steps of your data aggregation tasks—*Split-Apply-Combine* process, which is similar to *MapReduce* programming model for distributed computing systems.



Data Aggregation(cont.)

- Consider a simple example to get average *mpg* for different *hp* by applying the Split-Apply-Combine process to the *mtcars* data.

```
# “Split” mtcars by different cutpoints of horsepower(hp)
kv1 = split(mtcars,
            cut(mtcars$hp, breaks = c(0,100,200,Inf)));

# “Apply” mean() for each key-value pair
kv2 = lapply(kv1, FUN=function(x) return( mean(x$mpg)) );

# “Combine” the result
rt = data.frame(cbind(kv2)); colnames(rt) = 'average_mpg';

# Equivalent to the above statements
aggregate(mtcars$mpg,by = list(hp = cut(mtcars$hp,
          breaks = c(0,100,200,Inf))), FUN=mean)
```

- Many R aggregation functions can be used to simplify this process. You can also create your own process for more complicated tasks.

Data Aggregation(cont.)

- Let's see a bit more challenge but practical example of the process—fit models for different pieces of a dataset. It can be easily achieved by using *plyr::dlply()*. Take **mtcars** dataset again as example. Consider fitting simple linear models ***mpg ~ wt*** for different ***am*** (transmission type, 1=manual, 0 = automatic) .

```
library(plyr)
two_lm = dlply(.data = mtcars[,c('mpg','wt','am')],
                .variable = 'am',
                .fun=function(x) lm(x$mpg ~ x$wt))

 summary(two_lm$'1'); # model for am=1, manual
 summary(two_lm$'0'); # model for am=0, automatic
 # Plot it! Remember to load package "ggplot2"
 ggplot(data=mtcars,mapping = aes(x = wt, y=mpg)) +
  geom_point() + stat_smooth(method=lm) +
  facet_grid( ~ am, labeller = label_both);
```

- Also check out package *plyr* for more information.

Data Aggregation(cont.)

- The Split-Apply-Combine pattern is very common. Recognize and abstract such patterns in your daily data management tasks!

- This pattern provides us a easy way to simplify our code and parallelize computations.

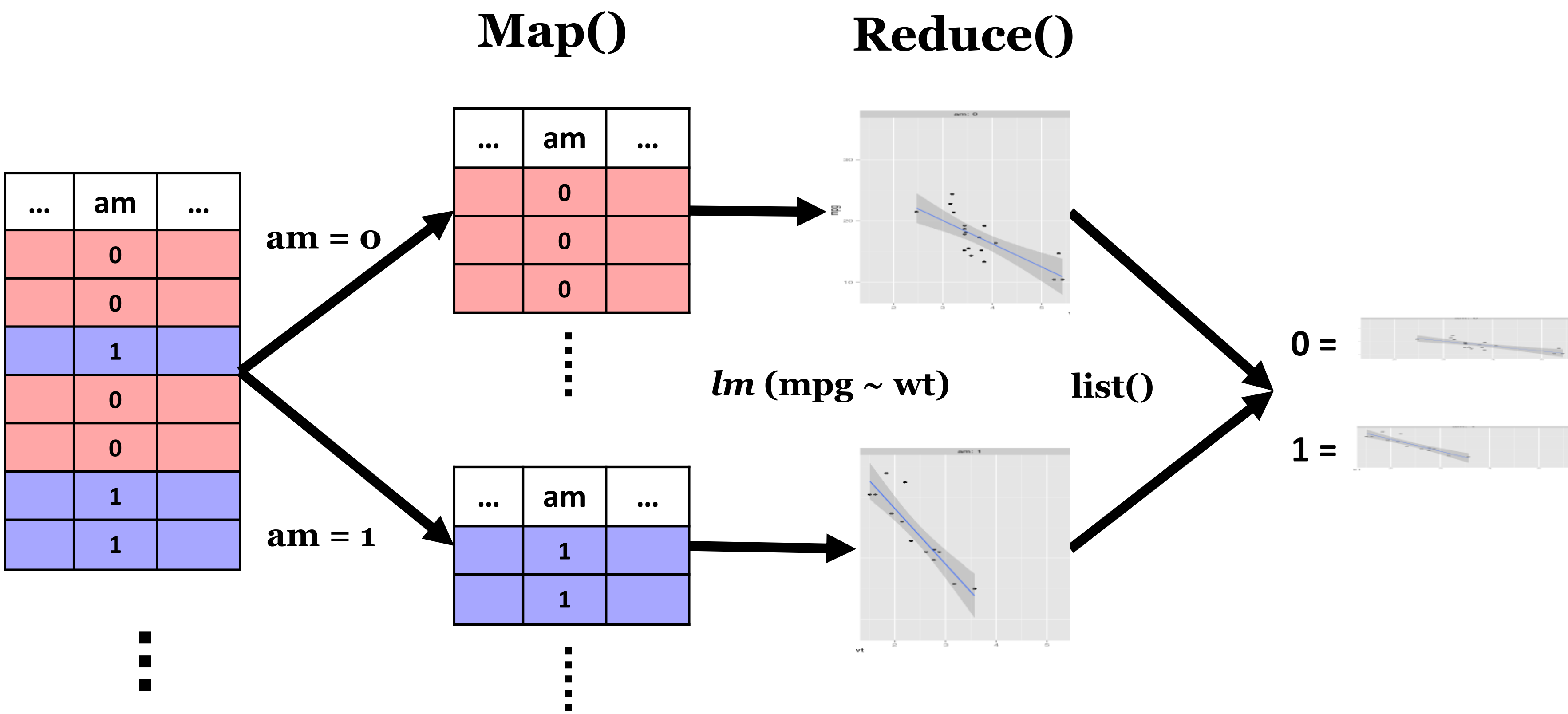
- Again, DO NOT use loop unless you really have to! E.g. in cases that the output of one iteration only depends on the result of previous iteration(s).

Think "Big"!

- We can do the same task in MapReduce manner on *Hadoop Distributed File System*(HDFS) with *RHadoop*. Let's put our *mtcars* on a given HDFS.

```
library(rmr2); # Load package rmr2 of RHadoop
mtcars_dfs = to.dfs(mtcars); # Put mtcars on HDFS
mtcars_fit = from.dfs( mapreduce( input = mtcars_dfs,
    map = function(k, v){
      return(keyval(key =  v$am, val = v )); # Split by "am"
    },
    reduce = function(k, v){
      return(keyval(key=k, val=list(lm(mpg ~ wt, data=v))));
    } )
);
 summary(mtcars_fit$val[[1]]); # model for am=0, automatic
 summary(mtcars_fit$val[[2]]); # model for am=1, manual
```

Think "Big"! (cont.)



Split by a key (value of "am")

Apply *lm*() to fit linear model for each subset of *mtcars* dataset. Two lm objects are created.

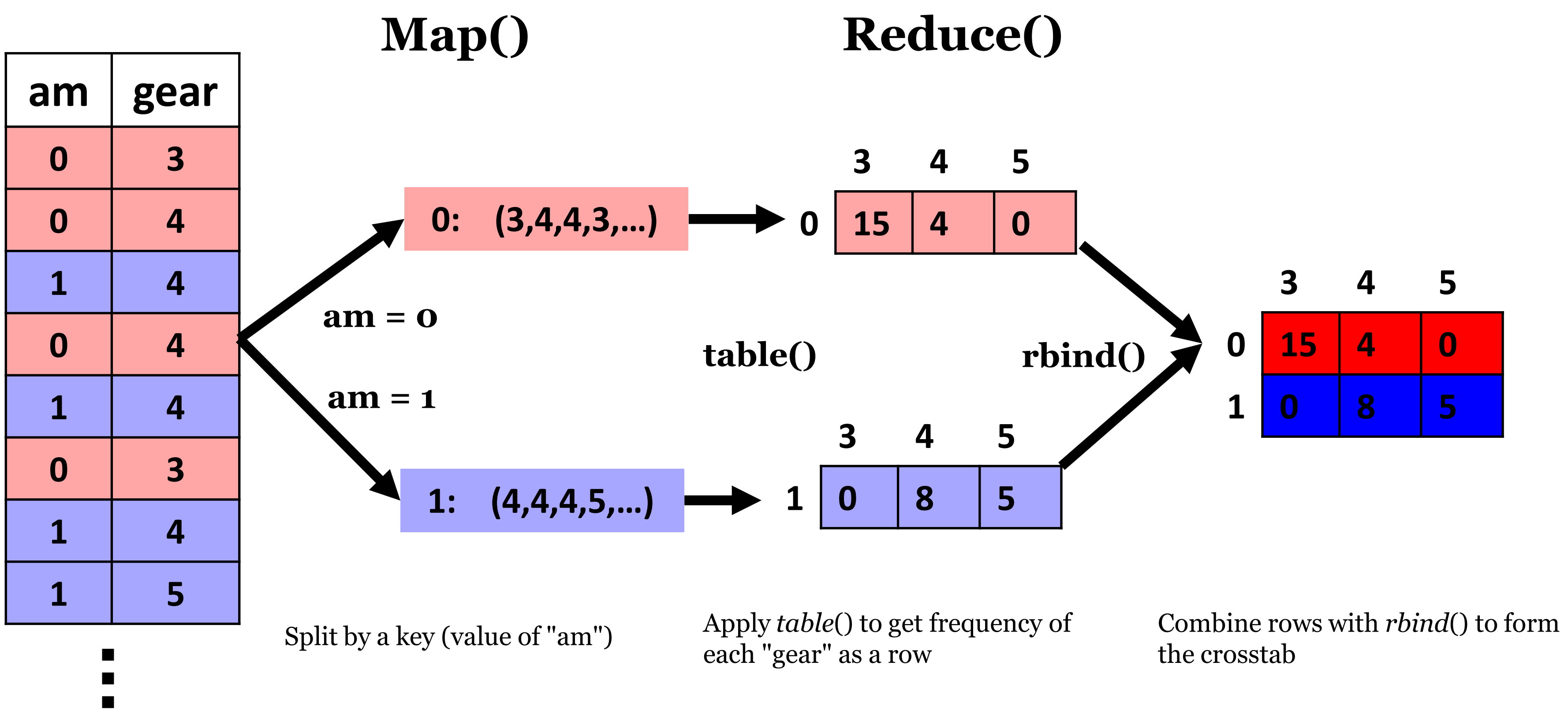
Combine two lm objects with *list* () then return the list object

Think "Big"! (cont.)

- Let's check out another example to get a crosstab of variable "x" by "y" similar to what we did with *table*() on a single machine. But now we assume we have a big dataset (say *mtcars* is big) on HDFS.

```
mtcars_dfs = to.dfs(mtcars); # Put mtcars on HDFS
# a function to get crosstab, "x" by "y"
crosstab_MR = function(dfs_data, x, y, ylevels){
  mapreduce( input = dfs_data,
    map = function(k, v){
      # Split by "x" values as the keys
      return(keyval(key = v[,x], val = v[, y] ));
    },
    reduce = function(k, v){
      tab = rbind(table(factor(v,levels=ylevels) ));
      rownames(tab) = k;
      return(keyval(key=k, val=tab));
})}
from.dfs(crosstab_MR(mtcars_dfs, x = 'am', y = 'gear',
         ylevels = c(3,4,5))); # get result from HDFS
```

Think "Big"! (cont.)

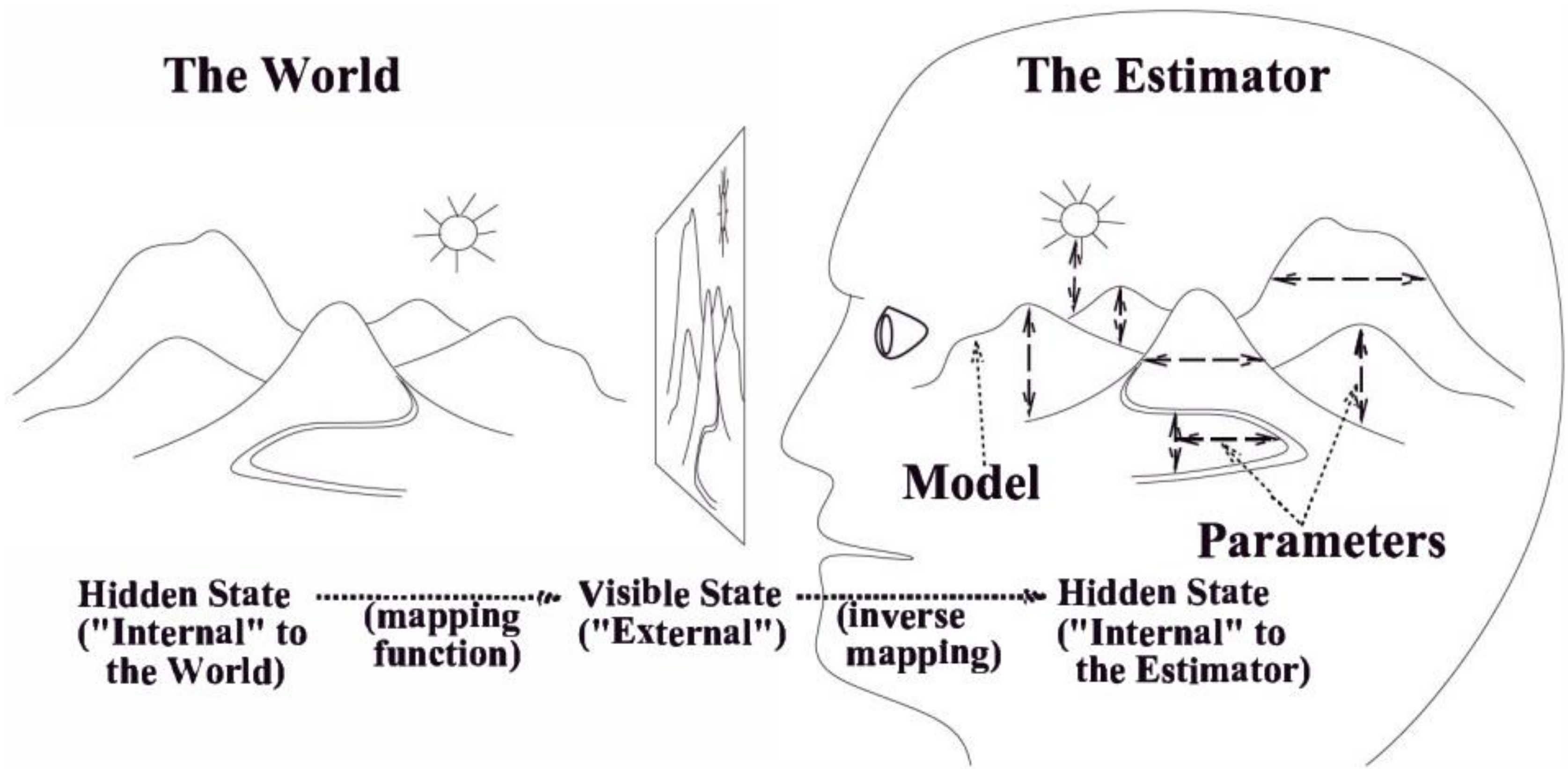


- We'll discuss more MapReduce design patterns later!

"Essentially, all models are wrong,
but some are useful."

— *George E. P. Box*

Model Thinking



Source: R. P. Rao, "*An optimal estimation approach to visual perception and learning*," Vision Res., vol. 39, no. 11, pp. 1963–1989, 1999.

Model Thinking(cont.)

- The *visible states* are observable conditions or certain outcome of the system, whereas the *hidden states* are those latent patterns that may contribute to the changes of the visible outcomes.

- Note that we only have *partial information* (i.e. variables) about the dynamic system or phenomenon we observe, simply because we don't know everything. We're not the God!

- We may end up realizing that "*Big Data" just means more pieces of a bigger puzzle*. Trying to define connection (e.g. causality) is human-nature, but somehow unrealistic and might not be necessary in the age of Big Data.

Model Thinking(cont.)

- By the "models" here, we mean mathematical model that describes a system using mathematical concepts. In R and most scientific computing software, a model is usually represented by an equation of both sides—outcomes and predictors.

- Consider a general linear model $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \varepsilon$, for example. This model $y = f(X_1, X_2)$ is coded in following R formula, as

$$\boldsymbol{y} \sim \boldsymbol{X_1} + \boldsymbol{X_2}$$

- Note that R uses "~" instead of "=" found in other data analysis or Statistics software.

R Formula

- Most of model fitting functions in R accept the following compact symbolic form of model definitions:

Target/Response Variables ~ Predictors/Explanatory Variables

+	Add variable(s)
-	Delete variable(s)
:	Add interaction term(s) among variables
*****	Add variable(s) and also include interaction term(s)
^	Add interactions up to a specified degree.
I()	Treat variables in function I( ) AS IS

R Formula(cont.)

- Here are some examples:

y ~ a + b + c	Three explanatory variables
y ~ a + b + a : b	Two explanatory variables with an interaction
y ~ a + (b * c)	Equivalent to **y ~ a + b + c + b:c**
y ~ a * b * c – a:b:c	Main effects plus 2-way interactions Equivalent to **y ~ (a + b + c)^2**
y ~ . – z	All explanatory variables except **z**
y ~ a + b + I(a * b)	Two explanatory variables and a new variable that multiplies these two variables

- Enter *?formula* in RStudio Console for more information

Try It!

- Load the pre-installed dataset *mtcars*. Use function *lm*() to fit linear models, *mpg = f(wt, am, …)*. Specify interaction terms if you'd like. Does car weight(wt) and/or transmission type (am) have impact on its MPG?

```
data(mtcars);
```

- Load the pre-installed dataset *Titanic*. Use function *glm*() to fit logit models with the variables you'd like. Use *epiDisplay::logistic.display()* to display the result. Did you see anything interesting?

```
data(Titanic);
install.packages("epiDisplay");
```