자기가 원하는 제목

# R basic

By the end of this class, you will be able to do arithmetic using R use variable, matrices, vector, dataframe in R explain data types in R

The credit of this lecture note is on Datacamp R courses, Coursera R programming courses, Kosuke Imai (2018), and Bogang Jun. ### 위의 insert에서 r을 누르면 회색영역 추가가능

## 1) Arithmetic

R uses =, -, \* like a caculator

10-1

## [1] 9

3\*5

## [1] 15

1+2

## [1] 3

You can also use / to divide

6/3

## [1] 2

10/5

## [1] 2

Like on a calculator, parentheses can be used to prioritize calculation

(2+1)\*10

## [1] 30

In R, ^ and \*\* are exponentiation operators. They raise the number on the left to the power of the number on the right: e.g. 3^2 == 9

2^3

## [1] 8

To find the square root of certain number, we need to use sqrt() function, while abs() function is used for taking the absolute value.

sqrt(9)

## [1] 3

abs(-9)

## [1] 9

The modulo operator %% returns the remainder of a division.

7%%2

## [1] 1

R ignores lines starting with #, so they can be used to add comments to your code.

## 2) R variables

Variables allow you to store value for later use. For example, to store the value 10 in the variable x, you would do: x <-10

age <- 23  
 age

## [1] 23

To output the value of a variable to the screen, you type the variable name.

pi <- 3.14  
 pi

## [1] 3.14

Storing a value in a variable is called assignment. You must assign a value to a variable before you can use it.

Variables can be used in operations anywhere you would use their values

income <- 100  
 costs <- 20  
 profit <- income - costs  
 profit

## [1] 80

If you assign a new value to an existing variable, it will replace the previous value

age <- 17  
 age <- 18  
 age

## [1] 18

## 3) Data types

Every value has a data type, which classifies the value. For example, name are character data but ages are numbers

R’s basic data types are character, numeric, integer, complex, and logical. R’s basic data structures include the vector, list, matrix, data frame, and factors. Objects may have attributes, such as name, dimension, and class.

Letters and words like “r”, “data”, or “analysis” (note the quotes) are called strings, and have the character data type.

You can use the class() function to get the data type of a value or a variable

class("HelloWorld")

## [1] "character"

You can use the str() function as well.

str("HelloWorld")

## chr "HelloWorld"

str(92)

## num 92

str(-1)

## num -1

str(1.23)

## num 1.23

Integers (whole numbers like 3) and numbers with decimal places like 3.14 have the numeric data type

n<- 3.14  
 class(n)

## [1] "numeric"

You often need to know if things are true or false: e.g. is it Friday? These can be represented by “Boolean”" values.

In R, TRUE and FALSE represent truthness and falseness, respectively. They have the logical data type, and have to be uppercase.

Why does 2 + ‘2’ cause an error? ’’ means words!(Chr)

## 4) Vectors

What’s the typical data structure in data analysis? Columns represent vectors (vaiables) and rows represent each observation.

Using a vector, we can store multiple values. You create them using the fuction c().

c(1,2,3)

## [1] 1 2 3

R prints the elements in a vector separated by spaces. The [1] is not part of the vector, it is only part of how R disply it.

c() combines the elements separated by commas into a vector.

You can assign a vector to a variable and print it. R will display the values separated by spaces.

v <- c(1,2,3)  
 v

## [1] 1 2 3

Vectors may contain character or Boolean values instead of numbers.

v <- c("x", "y", "z")  
 v

## [1] "x" "y" "z"

v<- c(TRUE, FALSE, TRUE)  
 v

## [1] TRUE FALSE TRUE

Within a single vector, all elements have the same data type.

The elements of a vector can have names: these get printed, too. You add names to a vector using another vector!

Note that when a vector has names, R doesn’t print them wih the preceding [1]

value <- c(1,2,3)  
 title <- c("x", "y", "z")  
 names(value) <- title  
 value

## x y z   
## 1 2 3

The names of a vector can also be printed on their own.

names(value)

## [1] "x" "y" "z"

Performing calculations with vectors You can fo arithmetic when creating vectors, and the result will be stored in the vector

v <- c(1+1, 2+2, 3+3)  
 v

## [1] 2 4 6

If you add two vectors with +, R will add the elements of the vectors element-wise.

a <- c(1,2,3)  
 b <- c(4,5,6)  
 a+b

## [1] 5 7 9

You can store the result of adding two vectors in a new vector

a <- c(1,2,3)  
 b <- c(4,5,6)  
 c <- a+b  
 c

## [1] 5 7 9

sum() is a function that will take the sum of everything within its parenthese.

v<- c(5,6,7)  
 sum(v)

## [1] 18

length() is a function that will take the number of elements.

length(v)

## [1] 3

### Subsetting vectors \*\*\*\*\*\*\*\*

You can pinpoint an element in a vector by its index. The first element is at index 1, the second is at 2, and so on.

You can use the index surrounded by square brackets([]) to extract an element from a vector

a <- c(2,3,4)  
a[1]

## [1] 2

a

## [1] 2 3 4

You can also get multiple elements from a vector by putting a vector of indices within the brackets. It’s vectors all the way down.

a[2]

## [1] 3

a[1:3]

## [1] 2 3 4

a[2:3]

## [1] 3 4

v <- c(6,7,8,9,10)  
v[c(1,2)]

## [1] 6 7

v[1:2]

## [1] 6 7

v[c(1,5,2)]

## [1] 6 10 7

The element you select correspond directrly to the indices you use, in whatever order you specify.

v[c(5,3,4)]

## [1] 10 8 9

You can save the results of subsetting a vector as a new vector

a <- c("white", "red", "yellow")  
b <- a[c(2,3)]  
b

## [1] "red" "yellow"

c <- a[1:2]  
c

## [1] "white" "red"

R has a shortcut for creating a vector of sequential integers. Typing 1:4 create a vector of the value 1 2 3 4

1:4

## [1] 1 2 3 4

2:10

## [1] 2 3 4 5 6 7 8 9 10

3:30

## [1] 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27  
## [26] 28 29 30

a <- 1:5  
a

## [1] 1 2 3 4 5

c(1:4)

## [1] 1 2 3 4

c(1,2,3,4)

## [1] 1 2 3 4

This is creating the same result with c(1,2,3,4,5) and c(1:5)

Subseting a vector using this shortcut

c(9,10,7,1)[2:3]

## [1] 10 7

c(9,10,7,1)[c(2,4)]

## [1] 10 1

c(9,10,7,1)[c(2:4)]

## [1] 10 7 1

You can subset a vector using its names.

v <- c(1,2,3,4)  
names(v) <- c("cat", "dog", "bird", "tiger")  
v

## cat dog bird tiger   
## 1 2 3 4

v[2]

## dog   
## 2

v["cat"]

## cat   
## 1

v[1:2]

## cat dog   
## 1 2

Previously you took the sum of a vector with sum(). mean() calculates the sum of the elements divided by how many they are.

v<- c(2,3,4)  
sum(v)

## [1] 9

mean(v)

## [1] 3

## 5) Comparing values in R

R has comparison operators. For example, you can check if one value is equal (==) or not equal (!=) to another.

When you compare two numbers or strings, you get a Boolean.

a=3  
a

## [1] 3

a==4

## [1] FALSE

a!=4

## [1] TRUE

4>3

## [1] TRUE

4<3

## [1] FALSE

4>=3

## [1] TRUE

4==4

## [1] TRUE

"book" != "cook"

## [1] TRUE

"book" == "book"

## [1] TRUE

"book" = 3  
book

## [1] 3

<= checks if a value is less than or equal to another, >= checks if it is greater than or equal another.

5>=5

## [1] TRUE

4<=3

## [1] FALSE

8<=9

## [1] TRUE

You can use these comparison operators to compare each element in a vector to some other value.

v <- c("cat", "dog", "rat")  
v== "cat"

## [1] TRUE FALSE FALSE

Remember how you subset vectors using vectors of indices or names. You can also subset using Booleans.

v <- c("small", "medium", "large")  
big <- v != "small"  
big

## [1] FALSE TRUE TRUE

v[big]

## [1] "medium" "large"

v <- c(5,6,7,8)  
mean(v[1:3]) >5

## [1] TRUE

mean(v[c(3,4)])

## [1] 7.5

c.f.error message appears when mean(v[1,2,3])

Using what you’ve learn, you can add two vectors and find when the resulting elements meet some threshold.

x <- c(10,1,11,18)  
y <- c(6, 10, 2,0)  
total <- x+y  
sel <- total >= 16  
sel

## [1] TRUE FALSE FALSE TRUE

total[sel]

## [1] 16 18

x[sel] ####sel은total >=16인 트 패패 트 결과물을 받았으므로 x[sel]은 트패패트의 결과물인 10과 18만 가져옴.

## [1] 10 18

## 6) matrices ##매트릭스는 항상 같은 타입의 데이터만 사용가능

Creating matrices You leant how to use c() for creating vectors. c() can also combine multiple vectors into one.

a <- c(1,2,3)  
b <- c(4,5,6)  
c(a,b)

## [1] 1 2 3 4 5 6

A matrix stores data in rows and columns, like table. You can create a matrix using the matrix() function

matrix(1:4,nrow=2)

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

matrix(1:4,nrow=2,byrow = TRUE)

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

You create a matrix with the matrix() function. nrow is an argument that tells matrix() how many rows there should be.

matrix(c("a","b"),nrow=2)

## [,1]  
## [1,] "a"   
## [2,] "b"

By defualt, matrix() fills in values from top to bottom. So it will fill in the first column,then the second, and so on.

matrix(1:6,nrow=3)

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

Again, by defualt, matrix() fills in values top to bottom, but you can choose to fill them in left to right using the byrow argument.

matrix(1:6,nrow=3, byrow = TRUE)

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

The “byrow”" argument to matrix() can be FALSE (the default) or TRUE.

matrix(c(2,4,6,9),byrow = TRUE, nrow = 2)

## [,1] [,2]  
## [1,] 2 4  
## [2,] 6 9

matrix(c(2,4,6,9),byrow = FALSE, nrow = 2)

## [,1] [,2]  
## [1,] 2 6  
## [2,] 4 9

To hep you remember what’s stord in your data, you can give the rows and columns names.

m <- matrix(c(20,15),nrow=2)  
m

## [,1]  
## [1,] 20  
## [2,] 15

rownames(m) <- c("yong", "sang")  
m

## [,1]  
## yong 20  
## sang 15

colnames(m) <- c("age")  
m

## age  
## yong 20  
## sang 15

### Combining matrices

Remember that matrices have rows and columns You can do some useful calculations on matrices, like finding the sum of each row with the rowSums() function

m <- matrix(c(20:23),nrow=2)  
m

## [,1] [,2]  
## [1,] 20 22  
## [2,] 21 23

rowSums() calculates the sum of each row, and colSum() calculates the sum of each column. ##S는 무조건 대문자

rowSums(m)

## [1] 42 44

colSums(m)

## [1] 41 45

Remember that calling c() on multiple vectors creates a new vector.

c(10:12,4:5,c(3,9))

## [1] 10 11 12 4 5 3 9

Just like you can combine multiple vectors into one, you can also combine matrices. rbind() stacks the rows from one matrix onto another.

m <- matrix(1:3,nrow=1)  
m

## [,1] [,2] [,3]  
## [1,] 1 2 3

n <- matrix(4:6,nrow=1)  
n

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

rbind(m,n)

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

###rbind = rowbind, cbind = colbind dz?

cbind() pastes the columns of two matrices together.

m <- matrix(1:3, nrow= 3)  
m

## [,1]  
## [1,] 1  
## [2,] 2  
## [3,] 3

n <- matrix(4:6, nrow =3)  
n

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

cbind(m,n)

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

cbind(n,m)

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

rbind() and cbind() can also combine multiple vectors into a matrix.

v <- c("a", "b", "c")  
v2 <- c("d", "e", "f")  
cbind(v,v2)

## v v2   
## [1,] "a" "d"  
## [2,] "b" "e"  
## [3,] "c" "f"

rbind(v,v2)

## [,1] [,2] [,3]  
## v "a" "b" "c"   
## v2 "d" "e" "f"

Recall that calling c() on two vectors creates a single vector.

v <- c(4, 6, 32, 55)  
m <- matrix(v,nrow=2)  
m

## [,1] [,2]  
## [1,] 4 32  
## [2,] 6 55

colSums(m)

## [1] 10 87

rbind(m,colSums(m))

## [,1] [,2]  
## [1,] 4 32  
## [2,] 6 55  
## [3,] 10 87

### Subsetting matrices

You can use [] to subset elements from a matrix. mat[1,2] selects the first row and second column.

m <- matrix(1:6, nrow=2)  
m

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

m[ 1 , 1]

## [1] 1

m[1,2]

## [1] 3

m[2,3]

## [1] 6

You can also use slicing to select elements from a matrix. mat[1:3, 2:4] selects data from rows 1,2,3, and cols 2,3,4.

m <- matrix(1:9, byrow = TRUE, nrow=3 )  
m

## [,1] [,2] [,3]  
## [1,] 1 2 3  
## [2,] 4 5 6  
## [3,] 7 8 9

m[1:2, 1:2]

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

m[1:3,1:2]

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

m[c(1,3),3]

## [1] 3 9

m[c(1,2), 1]

## [1] 1 4

m[c(2,3), 2]

## [1] 5 8

m[c(1,3), 1:3]

## [,1] [,2] [,3]  
## [1,] 1 2 3  
## [2,] 7 8 9

m[c(1,3), c(1,3)]

## [,1] [,2]  
## [1,] 1 3  
## [2,] 7 9

m[c(3,1),c(1,3)]

## [,1] [,2]  
## [1,] 7 9  
## [2,] 1 3

m[c(3,1),c(3,1)]

## [,1] [,2]  
## [1,] 9 7  
## [2,] 3 1

m[c(3,1),c(2,3)]

## [,1] [,2]  
## [1,] 8 9  
## [2,] 2 3

m[c(3),3]

## [1] 9

To select all elements of a matrix row or col, leave out the number before or after the comma. mat[,3] selects all elements of the third column.

m[,3]

## [1] 3 6 9

m[1,]

## [1] 1 2 3

Performing calculations and srithmetic with metrices Like with vectors, yu can use the arithmetric operators element wise on matrices: to add 3 to every element: mat +3.

m <- matrix(1:4, nrow=2,)  
m

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

m + 3

## [,1] [,2]  
## [1,] 4 6  
## [2,] 5 7

m\*2

## [,1] [,2]  
## [1,] 2 6  
## [2,] 4 8

m/2

## [,1] [,2]  
## [1,] 0.5 1.5  
## [2,] 1.0 2.0

m \* m

## [,1] [,2]  
## [1,] 1 9  
## [2,] 4 16

m%\*%m

## [,1] [,2]  
## [1,] 7 15  
## [2,] 10 22

You can also use operators on entire matrices. m1 \* m2 will multiply element in m1 by the corresponding element in m2.

m1 <- matrix(1:3, byrow = TRUE , nrow=1)  
m1

## [,1] [,2] [,3]  
## [1,] 1 2 3

m2 <- matrix(4:6,byrow = TRUE , nrow=1 )  
m2

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

m1 + m1

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

Just like with vectors, you can use funtions on matrices. mean() will get the average of all values in a matrix.

m <- matrix(1:4 , byrow= TRUE, nrow = 2)  
m

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

avg <- mean(m)  
avg

## [1] 2.5

m <- matrix(1:4, nrow=1)  
m

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

m[c(2,3)]

## [1] 2 3

mean(m[c(2,3)])

## [1] 2.5

## 7) Factors

As you have learned, vector can contain character strings. In R, factor vectors store data that falls into a finite number of categories. They are printed without quotation marks.

v <- c("a", "b", "b")  
class(v)

## [1] "character"

v2 <- factor(v)  
v2

## [1] a b b  
## Levels: a b

v

## [1] "a" "b" "b"

class(v2)

## [1] "factor"

The levels of a factor are the possible categories, and are printed along with the data.

v <- c("a", "b", "c", "ab", "ab","c")  
factor(v)

## [1] a b c ab ab c   
## Levels: a ab b c

When R prints factors it displays each vector element first, followed by the levels in the factor in alphabetical order.

v <- c("red", "blue", "red")  
factor(v)

## [1] red blue red   
## Levels: blue red

Factors store categorical data: data that has a finite number of distict categories, such as gender, country, race, or blood type.

R automatically sets the levels of the factor in alphabetical order. You can view them with levels().

v <- c("z","x","y","x")  
factor(v)

## [1] z x y x  
## Levels: x y z

levels(factor(v))

## [1] "x" "y" "z"

When you created factors with factor(), R automatically picked the levels. You can rename them if you want.

x

## [1] 10 1 11 18

x

## [1] 10 1 11 18

When you change the levels of a factor, it lso changes how the data is printed to use the new levels.

x

## [1] 10 1 11 18

Summarizing categorical variables Remember that R prints character vectors and factor vectors differently.

v <- c("a","b","c","c")  
f <- factor(v)  
summary(f)

## a b c   
## 1 1 2

f

## [1] a b c c  
## Levels: a b c

table(f)

## f  
## a b c   
## 1 1 2

Storing data in a factor can let you do useful things, like see the counts of each category with the summary() function.

You may be wondering what happens when you call summary() on a character vector.

x

## [1] 10 1 11 18

You can subset a factor with square brackets like you did with other vectors. When you subset a factor, R still displays all of the levels.

x

## [1] 10 1 11 18

You can compare strings alphabetically: “a is less than”b.

x

## [1] 10 1 11 18

The factors you’ve seen have no ranking of the levels, so if you compare them you’ll get NA and a warning.

x

## [1] 10 1 11 18

### Ordered factors

R can’t compare factors without an implied order. For example, R doesn’t know if “blue” is greater tha “red”.

Nominal variables represent categories of data that are unordered, whereas ordinary variables have an inherent order or ranking. (Letter grades: A, B, C… or Ages: young, teenage, old..)

R represents ordinal variables as ordered factors, using the ordered argument to factor(). Note the output: R orders alphabetically.

To explicily tell R the levels of a factor, you can add the levels argument to factor(). Note the output!

If you specify both ordered = TRUE and levels, R uses the order of levels.

So, to control the order of your ordered factors, you need to use the ordered and levels arguments.

## 8) Data Frames

We have already learned how to created matrices. Data Frames are another way to put data in tables. data.frame(a = 1:3) makes a data frame with a column named a. ## matrix는 모두 데이터가 같은 종류여야하지만 data.frame은 각각 다른 열로 섞일 수 있음.

d <- data.frame(a = 1:3, b= 4:6)  
d # do it on console

## a b  
## 1 1 4  
## 2 2 5  
## 3 3 6

d

## a b  
## 1 1 4  
## 2 2 5  
## 3 3 6

Each column in a data frame is a vector.

a <- c(1,2,3)  
b <- c(4,5,6)  
d <- data.frame(a,b)  
d

## a b  
## 1 1 4  
## 2 2 5  
## 3 3 6

Unlike matrices, data frames can have different types of data.

d <- data.frame(a = 1:2, b = c("x","y"), c= c(FALSE, TRUE))  
d

## a b c  
## 1 1 x FALSE  
## 2 2 y TRUE

###but within a column, all values must be the same type.

When you have a large data, head() will let you view just the first few rows. (c.f. tail())

d <- data.frame(a= 20:49, b = 50:79)  
head(d)

## a b  
## 1 20 50  
## 2 21 51  
## 3 22 52  
## 4 23 53  
## 5 24 54  
## 6 25 55

tail(d)

## a b  
## 25 44 74  
## 26 45 75  
## 27 46 76  
## 28 47 77  
## 29 48 78  
## 30 49 79

str() is another useful function that will tell you about the structure of your dataset.

str(d)

## 'data.frame': 30 obs. of 2 variables:  
## $ a: int 20 21 22 23 24 25 26 27 28 29 ...  
## $ b: int 50 51 52 53 54 55 56 57 58 59 ...

### Subsetting data frames

Subsetting data frames with the same way for a matrix.

d <- data.frame(a = 1:3, b= 4:6)  
d

## a b  
## 1 1 4  
## 2 2 5  
## 3 3 6

d[2,2]

## [1] 5

d[,1]

## [1] 1 2 3

d[1,]

## a b  
## 1 1 4

d[1:2,1:2]

## a b  
## 1 1 4  
## 2 2 5

You can use : to select a range of rows or columns.

d[2:3,1:2]

## a b  
## 2 2 5  
## 3 3 6

d[1,1]

## [1] 1

If you leave out the row numbers, R will return all the rows. (for the same with columns.)

d[1:2,]

## a b  
## 1 1 4  
## 2 2 5

You can subset data using the name of columns.

d <- data.frame(a = 1:3, b= 4:6)  
d[2 ,"b" ]

## [1] 5

d[1,"b"]

## [1] 4

mean(d[1:2,"b"])

## [1] 4.5

There is a shortcut to extract a column from a data frame. d$col gets the column col from the data frame d.

d$a

## [1] 1 2 3

d$a[2]

## [1] 2

d[2, "a"]

## [1] 2

d[1 , "b"]

## [1] 4

d$b[1]

## [1] 4

Another way to subset rows is to use a vector of Booleans. R will keep the rows corresponding to TRUE and drop those corresponding to FALSE.

df <- data.frame(a= 1:5, b = 6:10)  
v <- c(FALSE , TRUE, TRUE, FALSE, FALSE)   
df2 <- df[ v, ]  
df2

## a b  
## 2 2 7  
## 3 3 8

df[v, "a"]

## [1] 2 3

df$a[v]

## [1] 2 3

df[1,"a"]

## [1] 1

The function subset() will also return a subset of a data frame. You can use a vector of Booleans to tell it which rows to keep.–> use ‘?subset’

d <- data.frame(a= 1:3, b=4:6)  
subset(d)

## a b  
## 1 1 4  
## 2 2 5  
## 3 3 6

subset(d, c(TRUE, TRUE, FALSE))

## a b  
## 1 1 4  
## 2 2 5

subset(d, c(TRUE, TRUE, FALSE),"a")

## a  
## 1 1  
## 2 2

subset(d, c(TRUE, TRUE, FALSE),)

## a b  
## 1 1 4  
## 2 2 5

d[c(TRUE, TRUE, FALSE), ]

## a b  
## 1 1 4  
## 2 2 5

Remember how you can compare a vector to a single value and get a vector of Booleans.

V <- 1:5  
V > 3

## [1] FALSE FALSE FALSE TRUE TRUE

You can use subset() with comparison operators to get the rows that meet a condition.

df <- data.frame(a = 1:5, b =6:10)  
subset(df,,a)

## a  
## 1 1  
## 2 2  
## 3 3  
## 4 4  
## 5 5

subset(df, a>3 )

## a b  
## 4 4 9  
## 5 5 10

subset(df, b>8 )

## a b  
## 4 4 9  
## 5 5 10

subset(df, b>8 ,"a")

## a  
## 4 4  
## 5 5

######\*

df[which(df$a>3),]

## a b  
## 4 4 9  
## 5 5 10

df[df$a>3, ]

## a b  
## 4 4 9  
## 5 5 10

df[c(4,5),]

## a b  
## 4 4 9  
## 5 5 10

df <- data.frame(x = 1:70, y = 71:140)  
subset(df,y == 75)

## x y  
## 5 5 75

df[df$y==75, ]

## x y  
## 5 5 75

subset(df, x >60)

## x y  
## 61 61 131  
## 62 62 132  
## 63 63 133  
## 64 64 134  
## 65 65 135  
## 66 66 136  
## 67 67 137  
## 68 68 138  
## 69 69 139  
## 70 70 140

df[ df$x> 60, ]

## x y  
## 61 61 131  
## 62 62 132  
## 63 63 133  
## 64 64 134  
## 65 65 135  
## 66 66 136  
## 67 67 137  
## 68 68 138  
## 69 69 139  
## 70 70 140

The order() function gives the indices of a vector that would rearrange it in sorted order. order(c(5,10, 1)) returns the vector [1] 3 1 2.

v <- c(2,7,1,0)  
order(v)

## [1] 4 3 1 2

Recall that order() returns the indices of a vector that would rearrange that vector into ascending order.

You can also use the indices provided by order() to reorder the vector.

v[order(v)]

## [1] 0 1 2 7

v[c(4,3,1,2)]

## [1] 0 1 2 7

Remember that df$col gives you a vector containing the column col from the data frame df.

a <- c(1,4,2,3)  
b <- 6:9  
d <- data.frame(a,b)  
d

## a b  
## 1 1 6  
## 2 4 7  
## 3 2 8  
## 4 3 9

order(d$a)

## [1] 1 3 4 2

d[order(d$a),]

## a b  
## 1 1 6  
## 3 2 8  
## 4 3 9  
## 2 4 7

You can save the results of ordering a vector as a new vector to make it easier to work with.

b <- c(5,2,7,3)  
d <- data.frame(a =1:4,b)  
pos <- order(d$b)  
pos

## [1] 2 4 1 3

d$b[pos]

## [1] 2 3 5 7

You can use order() to sort the rows of a data frame based on the values in a column. The row numbers stay attached to their original rows.

a <- c(2,1,3)  
b <- c("x", "y", "z")  
d <- data.frame(a,b)  
order(a)

## [1] 2 1 3

order(d$a)

## [1] 2 1 3

d[order(d$a),]

## a b  
## 2 1 y  
## 1 2 x  
## 3 3 z

a <- c(6,4,7,2,9)  
d <- data.frame(a, b = 1:5)  
order(d$a)

## [1] 4 2 1 3 5

d[order(d$a),]

## a b  
## 4 2 4  
## 2 4 2  
## 1 6 1  
## 3 7 3  
## 5 9 5

d[order(a),]

## a b  
## 4 2 4  
## 2 4 2  
## 1 6 1  
## 3 7 3  
## 5 9 5

subset(d, b >2,)

## a b  
## 3 7 3  
## 4 2 4  
## 5 9 5

#d[d$b>2,]  
d[order(d$a),]

## a b  
## 4 2 4  
## 2 4 2  
## 1 6 1  
## 3 7 3  
## 5 9 5

## 9) List

List can hold diffierent kinds of objects like vectors, matrices, and data frame. The output below is a list, created with the function list().

list(1:3, 4:6)

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

The elements in he list are given numbers like [[1]], [[2]], etc.

list(1:3, matrix(4:6,nrow=1))

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

list(3, "sandwich")

## [[1]]  
## [1] 3  
##   
## [[2]]  
## [1] "sandwich"

When you create a list, you can name the elements, and the names will be printed, preceded by $.

list(a = 1:3, b = matrix(4:6, nrow=1))

## $a  
## [1] 1 2 3  
##   
## $b  
## [,1] [,2] [,3]  
## [1,] 4 5 6

Recall that you can set the names of a vector using names(). The same works for lists.

l <- list(7:9, 10:12)  
names(l) <- c("x", "y")  
l

## $x  
## [1] 7 8 9  
##   
## $y  
## [1] 10 11 12

c() function can new elements to an existing list.

l <- list(BOb =6)  
c(l, Amy = 2)

## $BOb  
## [1] 6  
##   
## $Amy  
## [1] 2

l

## $BOb  
## [1] 6

l <- c(l, Amy = 2)  
l

## $BOb  
## [1] 6  
##   
## $Amy  
## [1] 2

l <- c(l, y= 3) #l은 1이 아니다.  
l

## $BOb  
## [1] 6  
##   
## $Amy  
## [1] 2  
##   
## $y  
## [1] 3

Vectors, matrices, data frames, ad lists are all data structures: way to store in R. Among these, lists are the super data structure: they can contain all the other kinds of data structures.

Remember that lists can contain lots of different things, and each element is given a number ([[1]]) or name.

l <-list(5, "cat", TRUE)  
names(l) <- c("x","y")  
l

## $x  
## [1] 5  
##   
## $y  
## [1] "cat"  
##   
## $<NA>  
## [1] TRUE

Lists can contain many different elements. To extract the first element from list a, write a[[1]].

d <- data.frame (x= 1:2, y= 3:4)  
m <- matrix(1:4, nrow=2)  
mylist <- list(d,m)  
mylist

## [[1]]  
## x y  
## 1 1 3  
## 2 2 4  
##   
## [[2]]  
## [,1] [,2]  
## [1,] 1 3  
## [2,] 2 4

mylist[2]

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

mylist[[2]]

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

Remember to use double square brackets ([[]]) to extract an element from the list.

d <- data.frame(a = 1:5)  
v <- 1:5  
mylist <- list(d,v)  
mylist[[2]]

## [1] 1 2 3 4 5

mylist[[1]]

## a  
## 1 1  
## 2 2  
## 3 3  
## 4 4  
## 5 5

If a list’s elements are named, you can put the quoted name of the element you want within the double square brackets.

l <- list(a = 1:3, b = 4:6)  
l[["a"]]

## [1] 1 2 3

l[[1]]

## [1] 1 2 3

m <- matrix(1:4, nrow=1)  
d <- data.frame(a = 1:2, b = c("cat","duck"))  
l <- list(m,df= d)  
l[["df"]]

## a b  
## 1 1 cat  
## 2 2 duck

You can also use the shortcut you learned with data frames to extract a single element from a list: listname$element.

l <- list(fruit = "apple", nums=5:7)  
l

## $fruit  
## [1] "apple"  
##   
## $nums  
## [1] 5 6 7

l$fruit

## [1] "apple"

l$nums

## [1] 5 6 7

You can even combine the subsetting techniques you’ve learned. To get the second value from the first element of list l, write l[[1]][2].

l <- list(a = 1:5, b= 6:10)  
l[["b"]][1]

## [1] 6

l[[1]][1]

## [1] 1

l[1][1]

## $a  
## [1] 1 2 3 4 5

x <- data.frame(a = 1:50, b = 51:100)  
l <- list(df = x, vec = c(TRUE, FALSE))  
l

## $df  
## a b  
## 1 1 51  
## 2 2 52  
## 3 3 53  
## 4 4 54  
## 5 5 55  
## 6 6 56  
## 7 7 57  
## 8 8 58  
## 9 9 59  
## 10 10 60  
## 11 11 61  
## 12 12 62  
## 13 13 63  
## 14 14 64  
## 15 15 65  
## 16 16 66  
## 17 17 67  
## 18 18 68  
## 19 19 69  
## 20 20 70  
## 21 21 71  
## 22 22 72  
## 23 23 73  
## 24 24 74  
## 25 25 75  
## 26 26 76  
## 27 27 77  
## 28 28 78  
## 29 29 79  
## 30 30 80  
## 31 31 81  
## 32 32 82  
## 33 33 83  
## 34 34 84  
## 35 35 85  
## 36 36 86  
## 37 37 87  
## 38 38 88  
## 39 39 89  
## 40 40 90  
## 41 41 91  
## 42 42 92  
## 43 43 93  
## 44 44 94  
## 45 45 95  
## 46 46 96  
## 47 47 97  
## 48 48 98  
## 49 49 99  
## 50 50 100  
##   
## $vec  
## [1] TRUE FALSE

l[["df"]][1]

## a  
## 1 1  
## 2 2  
## 3 3  
## 4 4  
## 5 5  
## 6 6  
## 7 7  
## 8 8  
## 9 9  
## 10 10  
## 11 11  
## 12 12  
## 13 13  
## 14 14  
## 15 15  
## 16 16  
## 17 17  
## 18 18  
## 19 19  
## 20 20  
## 21 21  
## 22 22  
## 23 23  
## 24 24  
## 25 25  
## 26 26  
## 27 27  
## 28 28  
## 29 29  
## 30 30  
## 31 31  
## 32 32  
## 33 33  
## 34 34  
## 35 35  
## 36 36  
## 37 37  
## 38 38  
## 39 39  
## 40 40  
## 41 41  
## 42 42  
## 43 43  
## 44 44  
## 45 45  
## 46 46  
## 47 47  
## 48 48  
## 49 49  
## 50 50

str(l)

## List of 2  
## $ df :'data.frame': 50 obs. of 2 variables:  
## ..$ a: int [1:50] 1 2 3 4 5 6 7 8 9 10 ...  
## ..$ b: int [1:50] 51 52 53 54 55 56 57 58 59 60 ...  
## $ vec: logi [1:2] TRUE FALSE

mydata <- l[[1]]  
mydata

## a b  
## 1 1 51  
## 2 2 52  
## 3 3 53  
## 4 4 54  
## 5 5 55  
## 6 6 56  
## 7 7 57  
## 8 8 58  
## 9 9 59  
## 10 10 60  
## 11 11 61  
## 12 12 62  
## 13 13 63  
## 14 14 64  
## 15 15 65  
## 16 16 66  
## 17 17 67  
## 18 18 68  
## 19 19 69  
## 20 20 70  
## 21 21 71  
## 22 22 72  
## 23 23 73  
## 24 24 74  
## 25 25 75  
## 26 26 76  
## 27 27 77  
## 28 28 78  
## 29 29 79  
## 30 30 80  
## 31 31 81  
## 32 32 82  
## 33 33 83  
## 34 34 84  
## 35 35 85  
## 36 36 86  
## 37 37 87  
## 38 38 88  
## 39 39 89  
## 40 40 90  
## 41 41 91  
## 42 42 92  
## 43 43 93  
## 44 44 94  
## 45 45 95  
## 46 46 96  
## 47 47 97  
## 48 48 98  
## 49 49 99  
## 50 50 100

l[["df"]][1,]

## a b  
## 1 1 51

#########현재의 x에는 a와b 두개가 있으므로 [[]][]하려면 [1,] ","을 써야함!

Do you remember the function str()? You can use it on lists to learn how many elements they contain, what types those elements are, etc.

str(l)

## List of 2  
## $ df :'data.frame': 50 obs. of 2 variables:  
## ..$ a: int [1:50] 1 2 3 4 5 6 7 8 9 10 ...  
## ..$ b: int [1:50] 51 52 53 54 55 56 57 58 59 60 ...  
## $ vec: logi [1:2] TRUE FALSE

l <- list(food = "pizza", nums = 4:7)  
str(l)

## List of 2  
## $ food: chr "pizza"  
## $ nums: int [1:4] 4 5 6 7

l[[2]][3]

## [1] 6