Part 1: Basics

[Introduction]

R Data Types, Arithmetic & Logical Operators with Example

3 classes

1. numeric

2. character

3. logical

Can create variables using <- or =

Vectors

Example 1:

```
# Numerical
vec_num <- c(1, 10, 49)
vec_num
```
NEED TO INCLUDE c()

Example 5:

In R, it is possible to slice a vector. In some occasion, we are interested in only the first five rows of a vector. We can use the [1:5] command to extract the value 1 to 5.

```
# Slice the first five rows of the vector
slice_vector <- c(1,2,3,4,5,6,7,8,9,10)
slice_vector[1:5]
# Faster way to create adjacent values
c(1:10)
```

Output:

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

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x	y
x & y	x AND y
isTRUE(x)	Test if X is TRUE

Example 1:

```
# Create a vector from 1 to 10
logical_vector <- c(1:10)
logical_vector>5
```

Output:

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

Example 2:

In the example below, we want to extract the values that only meet the condition 'is strictly superior to five'. For that, we can wrap the condition inside a square bracket precede by the vector containing the values.

```
# Print value strictly above 5
logical_vector[(logical_vector>5)]
```

```
> v
 [1]  1  2  3  4  5  6  7  8  9 10
> v>=5
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> v [(v>5)]
[1]  6  7  8  9 10
> v[v>5]
[1]  6  7  8  9 10
```

Example 3:

```
# Print 5 and 6
logical_vector <- c(1:10)
logical_vector[(logical_vector>4) & (logical_vector<7)]
```

Output:

```
## [1] 5 6
```

My example of printing only the number 6

```
> v[(v<7)&(v>5)]
[1] 6
```

[How to Create a Matrix in R]

We can create a matrix with the function matrix(). This function takes three arguments:

```
matrix(data, nrow, ncol, byrow = FALSE)
```

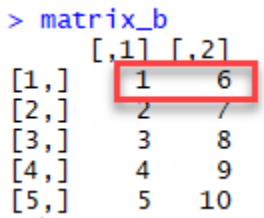
if you don't include byrow, then by default fills in column first like my 2nd example. So basically it's byrow=FALSE by default

```
> m = matrix(v, 5, 2)
> m
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

```
> m = matrix(v, 5, 2, byrow=TRUE)
> m
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
```

```
# Construct a matrix with 5 rows that contain the numbers 1 up to 10 and byrow =  FALSE
matrix_b <-matrix(1:10, byrow = FALSE, nrow = 5)
matrix_b
```

Output:




```
# Print dimension of the matrix with dim()
dim(matrix_a)
```

Output:

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

Add a Column to a Matrix with the cbind()

You can add a column to a matrix with the cbind() command. cbind() means column binding. cbind()can concatenate as many matrix or columns as specified. For example, our previous example created a 5x2 matrix. We concatenate a third column and verify the dimension is 5x3

Example:

```
# concatenate c(1:5) to the matrix_a
matrix_a1 <- cbind(matrix_a, c(1:5))
# Check the dimension
dim(matrix_a1)
```

Output:

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

Example:

```
matrix_a1
```

Output

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

```
> m2 = cbind (m,c(5:1))
> m2
     [,1] [,2] [,3]
[1,]    1    6    5
[2,]    2    7    4
[3,]    3    8    3
[4,]    4    9    2
[5,]    5   10    1
> m2 = cbind (m,c(1:5))
> m2
     [,1] [,2] [,3]
[1,]    1    6    1
```

```
[2,]    2    7    2
[3,]    3    8    3
[4,]    4    9    4
[5,]    5   10    5
```

This adds another column to the end with the specified vector x:y, where x is always at the top.

Example:

We can also add more than one column. Let's see the next sequence of number to the matrix_a2 matrix. The dimension of the new matrix will be 4x6 with number from 1 to 24.

```
matrix_c <-matrix(1:12, byrow = FALSE, ncol = 3)
matrix_d <- cbind(matrix_a2, matrix_c)
dim(matrix_d)
```

Output:

```
## [1] 4 6
> m = matrix (1:12, 4, 3, byrow = TRUE)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> m2 = matrix(13:24, 4, 3, byrow = TRUE)
> m2
     [,1] [,2] [,3]
[1,]   13   14   15
[2,]   16   17   18
[3,]   19   20   21
[4,]   22   23   24
 > m3 = cbind(m,m2)
 > m3
      [,1] [,2] [,3] [,4] [,5] [,6]
 [1,]    1    2    3   13   14   15
 [2,]    4    5    6   16   17   18
 [3,]    7    8    9   19   20   21
 [4,]   10   11   12   22   23   24
 > m4 = matrix(1:4, 4, byrow = TRUE)
 > m4
      [,1]
 [1,]    1
 [2,]    2
```

```
[3,]    3
[4,]    4
> m5 = cbind (m3,m4)
> m5
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    2    3   13   14   15    1
[2,]    4    5    6   16   17   18    2
[3,]    7    8    9   19   20   21    3
[4,]   10   11   12   22   23   24    4
```

Byrow = TRUE makes it so that the vectors are sorted by row

> **Slice a Matrix**

- matrix_c[1,2] selects the element at the first row and second column.
- matrix_c[1:3,2:3] results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3,
- matrix_c[,1] selects all elements of the first column.
- matrix_c[1,] selects all elements of the first row.

```
> m6 = m5[1,2]
> m6
[1] 2
```

This made m6 the value of m5's row 1 and column 2

If you only want the 1st and 7th column of m5 then do this

```
> m6=cbind(m5[,1],m5[,7])
> m6
     [,1] [,2]
[1,]    1    1
[2,]    4    2
[3,]    7    3
[4,]   10    4
```

[Factor in R: Categorical & Continuous Variables]

What is Factor in R?

Factors are variables in R which take on a limited number of different values; such variables are often referred to as categorical variables.

In a dataset, we can distinguish two types of variables: **categorical** and **continuous**.

- In a categorical variable, the value is limited and usually based on a particular finite group. For example, a categorical variable can be countries, year, gender, occupation.
- A continuous variable, however, can take any values, from integer to decimal. For example, we can have the revenue, price of a share, etc..

Categorical Variables

Syntax

```
factor(x = character(), levels, labels = levels, ordered = is.ordered(x))
```

Let's create a factor data frame.

```
# Create gender vector
gender_vector <- c("Male", "Female", "Female", "Male", "Male")
class(gender_vector)
# Convert gender_vector to a factor
factor_gender_vector <-factor(gender_vector)
class(factor_gender_vector)
```

Output:

```
## [1] "character"
## [1] "factor"
> gender = c("Male", "Female", "Female", "Male", "Male")
> gender
[1] "Male"   "Female" "Female" "Male"   "Male"
> class (gender)
[1] "character"

> gender_factor = factor(gender)
> gender_factor
[1] Male   Female Female Male   Male
Levels: Female Male
> class (gender_factor)
```

[1] "factor"

Basically, the only reason to change the categorical variables into factors is so that R can do it's thing. Otherwise, it won't do anything.

Nominal Categorical Variable

A categorical variable has several values but the order does not matter. For instance, male or female categorical variable do not have ordering.

Ordinal Categorical Variable

Ordinal categorical variables do have a natural ordering. We can specify the order, from the lowest to the highest with order = TRUE and highest to lowest with order = FALSE.

Example:

We can use summary to count the values for each factor.

```
# Create Ordinal categorical vector
day_vector <- c('evening', 'morning', 'afternoon', 'midday', 'midnight', 'evening')
# Convert `day_vector` to a factor with ordered level
factor_day <- factor(day_vector, order = TRUE, levels =c('morning', 'midday', 'afternoon',
'evening', 'midnight'))
# Print the new variable
factor_day
```

```
> nom_num = c("one", "two", "three")
> nom_num
[1] "one"   "two"   "three"
> ord_num = factor (nom_num, order = TRUE, levels =c("one", "two", "three"))
> ord_num
[1] one   two   three
Levels: one < two < three
> summary (ord_num)
  one   two three
    1     1     1
```

Do you convert string variables into factors so you can convert them into numerical values? Is this the only function of converting them to factors? **Sort of.**

[R Data Frame: Create, Append, Select, Subset]

What is a Data Frame?

A **data frame** is a list of vectors which are of equal length. A matrix contains only one type of data, while a data frame accepts different data types (numeric, character, factor, etc.).

```
# Create a, b, c, d variables
a <- c(10,20,30,40)
b <- c('book', 'pen', 'textbook', 'pencil_case')
c <- c(TRUE,FALSE,TRUE,FALSE)
d <- c(2.5, 8, 10, 7)
# Join the variables to create a data frame
df <- data.frame(a,b,c,d)
df
```

Output:

```
##   a      b c d
## 1  1       book  TRUE   2.5
## 2  2        pen  TRUE   8.0
## 3  3    textbook  TRUE  10.0
## 4  4 pencil_case FALSE   7.0
```

Naming the individual columns in data frame

```
# Name the data frame
names(df) <- c('ID', 'items', 'store', 'price')
df
```

Output:

```
##   ID       items store price
## 1 10        book  TRUE   2.5
## 2 20         pen FALSE   8.0
## 3 30    textbook  TRUE  10.0
## 4 40 pencil_case FALSE   7.0
# Print the structure
str(df)
```

Output:

```
## 'data.frame':    4 obs. of  4 variables:
```

```
##  $ ID   : num  10 20 30 40
##  $ items: Factor w/ 4 levels "book","pen","pencil_case",..: 1 2 4 3
##  $ store: logi  TRUE FALSE TRUE FALSE
##  $ price: num  2.5 8 10 7
## Select row 1 in column 2
df[1,2]
```

Output:

```
## [1] book
## Levels: book pen pencil_case textbook
## Select Rows 1 to 2
df[1:2,]
```

Output:

```
##   ID items store price
## 1 10  book  TRUE   2.5
## 2 20   pen FALSE   8.0
## Select Columns 1
df[,1]
```

Output:

```
## [1] 10 20 30 40
## Select Rows 1 to 3 and columns 3 to 4
df[1:3, 3:4]
```

Output:

```
##   store price
## 1  TRUE   2.5
## 2 FALSE   8.0
## 3  TRUE  10.0
```

Selecting specific column in data frame by their names

```
# Slice with columns name
df[, c('ID', 'store')]
```

Output:

```
##   ID store
## 1 10  TRUE
```

```
## 2 20 FALSE
## 3 30  TRUE
## 4 40 FALSE
```

Whats the differences between cbind and c function? Cbind assigns 1,2,3,4,5 to values or labels and c doesn't. Is that it? **C is only for vectors. Cbind can represent matrix and list, etc**

```
## Select row 1 in column 2
df[1,2]
```

If you want to select multiple non-adjacent columns and rows do this

```
> store_data [cbind(1,3),cbind(4,1)]
  Price ID
1   2.5  1
3  10.0  3
```

Pretty self-explanatory.
Remember always row first then column.

Output:

```
## [1] book
## Levels: book pen pencil_case textbook
```

Appending or adding new column to data frame

```
# Create a new vector
quantity <- c(10, 35, 40, 5)

# Add `quantity` to the `df` data frame
df$quantity <- quantity
df
```

Select a Column of a Data Frame

```
# Select the column ID
df$ID
```

Output:

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

Subset a Data Frame

We want to return only the items with price above 10, we can do:

```
# Select price above 5
subset(df, subset = price > 5)
```

Output:

```
ID       items store price
2 20         pen FALSE     8
3 30    textbook  TRUE    10
4 40 pencil_case FALSE     7
```

[List in R: Create, Select Elements with Examples]

Step 1) Create a Vector

```
# Vector with numeric from 1 up to 5
vect  <- 1:5
```

Step 2) Create a Matrices

```
# A 2x 5 matrix
mat  <- matrix(1:9, ncol = 5)
dim(mat)
```

Output:

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

Step 3) Create Data Frame

```
# select the 10th row of the built-in R data set EuStockMarkets
df <- EuStockMarkets[1:10,]
```

Step 4) Create a List

Now, we can put the three object into a list.

```
# Construct list with these vec, mat, and df:
my_list <- list(vect, mat, df)
my_list
```

Output:

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

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

## [[3]]
##         DAX    SMI    CAC   FTSE
##  [1,] 1628.75 1678.1 1772.8 2443.6
##  [2,] 1613.63 1688.5 1750.5 2460.2
```

```
##  [3,] 1606.51 1678.6 1718.0 2448.2
##  [4,] 1621.04 1684.1 1708.1 2470.4
##  [5,] 1618.16 1686.6 1723.1 2484.7
##  [6,] 1610.61 1671.6 1714.3 2466.8
##  [7,] 1630.75 1682.9 1734.5 2487.9
##  [8,] 1640.17 1703.6 1757.4 2508.4
##  [9,] 1635.47 1697.5 1754.0 2510.5
##  [10,] 1645.89 1716.3 1754.3 2497.4
```

Select Elements from List

Let's try to select the second items of the list named my_list, we use my_list[[2]]

```
# Print second element of the list
my_list[[2]]
```

Output:

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8    1
> df = EuStockMarkets [1:10,]
> vect  <- 1:5
> mat  <- matrix(1:10, ncol = 5)
> list(vect,mat,df)
[[1]]
[1] 1 2 3 4 5

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

[[3]]
         DAX    SMI    CAC
 [1,] 1628.75 1678.1 1772.8
 [2,] 1613.63 1688.5 1750.5
 [3,] 1606.51 1678.6 1718.0
 [4,] 1621.04 1684.1 1708.1
 [5,] 1618.16 1686.6 1723.1
 [6,] 1610.61 1671.6 1714.3
 [7,] 1630.75 1682.9 1734.5
 [8,] 1640.17 1703.6 1757.4
 [9,] 1635.47 1697.5 1754.0
[10,] 1645.89 1716.3 1754.3
```

```
       FTSE
 [1,] 2443.6
 [2,] 2460.2
 [3,] 2448.2
 [4,] 2470.4
 [5,] 2484.7
 [6,] 2466.8
 [7,] 2487.9
 [8,] 2508.4
 [9,] 2510.5
[10,] 2497.4
```

```
> my_list = list(vect,mat,df)
> my_list
[[1]]
[1] 1 2 3 4 5

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

[[3]]
         DAX    SMI    CAC
 [1,] 1628.75 1678.1 1772.8
 [2,] 1613.63 1688.5 1750.5
 [3,] 1606.51 1678.6 1718.0
 [4,] 1621.04 1684.1 1708.1
 [5,] 1618.16 1686.6 1723.1
 [6,] 1610.61 1671.6 1714.3
 [7,] 1630.75 1682.9 1734.5
 [8,] 1640.17 1703.6 1757.4
 [9,] 1635.47 1697.5 1754.0
[10,] 1645.89 1716.3 1754.3
       FTSE
 [1,] 2443.6
 [2,] 2460.2
 [3,] 2448.2
 [4,] 2470.4
 [5,] 2484.7
 [6,] 2466.8
 [7,] 2487.9
 [8,] 2508.4
 [9,] 2510.5
[10,] 2497.4
```

```
> my_list[2]
[[1]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Built-in Data Frame

Before to create our own data frame, we can have a look at the R data set available online. The prison dataset is a 714x5 dimension. We can get a quick look at the bottom of the data frame with **tail() function**. By analogy, **head() displays the top of the data frame**. You can specify the number of rows shown **with head (df, 5)**. We will learn more about the function read.csv() in future tutorial.

```
# Print the head of the data
PATH<-
'https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/wooldridge/prison.csv'
df <- read.csv(PATH)[1:5]
```

Output:

```
##   X state year govelec black
## 1 1     1   80       0 0.2560
## 2 2     1   81       0 0.2557
## 3 3     1   82       1 0.2554
## 4 4     1   83       0 0.2551
## 5 5     1   84       0 0.2548
```

[R Sort a Data Frame using Order()]

Syntax:

```
sort(x, decreasing = FALSE, na.last = TRUE):
```

Argument:

- **x**: A vector containing continuous or factor variable
- **decreasing**: Control for the order of the sort method. By default, decreasing is set to `FALSE`.
- **last**: Indicates whether the `NA` 's value should be put last or not

```
> sort(store_data$"Price")
[1]  2.5  7.0  8.0 10.0
```

If you want to order two sets of observations for the same variable you have to follow this format

- variable[order(variable$obs1,variable$obs2),]

```
data_frame <- tibble(
            c1 = rnorm(50, 5, 1.5),
            c2 = rnorm(50, 5, 1.5),
            c3 = rnorm(50, 5, 1.5),
            c4 = rnorm(50, 5, 1.5),
            c5 = rnorm(50, 5, 1.5)
)
```

Tibble by default sorts the observations by column with variable name at top of column

rnorm (n, mean, sd)

```
# Sort by c1
df <-data_frame[order(data_frame$c1),]


head(df,5)
```

head (variable, #) # chooses how many lines to show out of the total

```
order(…, na.last = TRUE, decreasing = FALSE,
      method = c("auto", "shell", "radix"))
```

Arguments

…

a sequence of numeric, complex, character or logical vectors, all of the same length, or a classed R object.

decreasing

logical. Should the sort order be increasing or decreasing? For the "radix" method, this can be a vector of length equal to the number of arguments in …. For the other methods, it must be length one.

na.last

for controlling the treatment of NAs. If TRUE, missing values in the data are put last; ifFALSE, they are put first; if NA, they are removed (see 'Note'.)

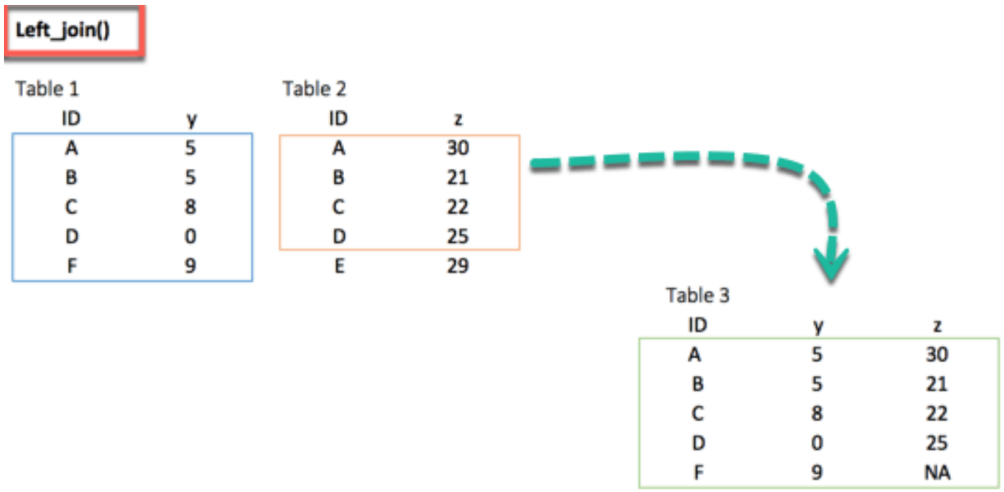
method

the method to be used: partial matches are allowed. The default ("auto") implies "radix"for short numeric vectors, integer vectors, logical vectors and factors. Otherwise, it implies "shell". For details of methods "shell", "quick", and "radix", see the help for sort.

[R Dplyr Tutorial: Data Manipulation(Join) & Cleaning(Spread)]

Merge with dplyr()

- Left_join()
- right_join()
- inner_join()
- full_join()




```
left_join(df_primary, df_secondary, by ='ID')
```

~"name" chooses the label for the set of observations (HAS TO BE INDENTED)

```
> df_primary <- tribble(
+   ~ID, ~y,
+   "A", 5,
+   "B", 5,
+   "C", 8,
+   "D", 0,
+   "F", 9
+ )
> df_primary
# A tibble: 5 x 2
  ID        y
  <chr> <dbl>
1 A         5
2 B         5
```

```
3 C        8
4 D        0
5 F        9
> df_primary=df1
Error: object 'df1' not found
> df1=df_primary
> df1
# A tibble: 5 x 2
  ID        y
  <chr> <dbl>
1 A         5
2 B         5
3 C         8
4 D         0
5 F         9
> df2= tribble(
+   ~ID, ~y,
+   "A", 30,
+   "B", 21,
+   "C", 22,
+   "D", 25,
+   "E", 29)
>
> df1
# A tibble: 5 x 2
  ID        y
  <chr> <dbl>
1 A         5
2 B         5
3 C         8
4 D         0
5 F         9
> df2
# A tibble: 5 x 2
  ID        y
  <chr> <dbl>
1 A        30
2 B        21
3 C        22
4 D        25
5 E        29
> left_join(df1,df2, by="ID")
# A tibble: 5 x 3
  ID      y.x   y.y
  <chr> <dbl> <dbl>
1 A         5    30
```

2	B	5	21
3	C	8	22
4	D	0	25
5	F	9	NA

By=”key variable”

- ***matches same key variables and combines them***

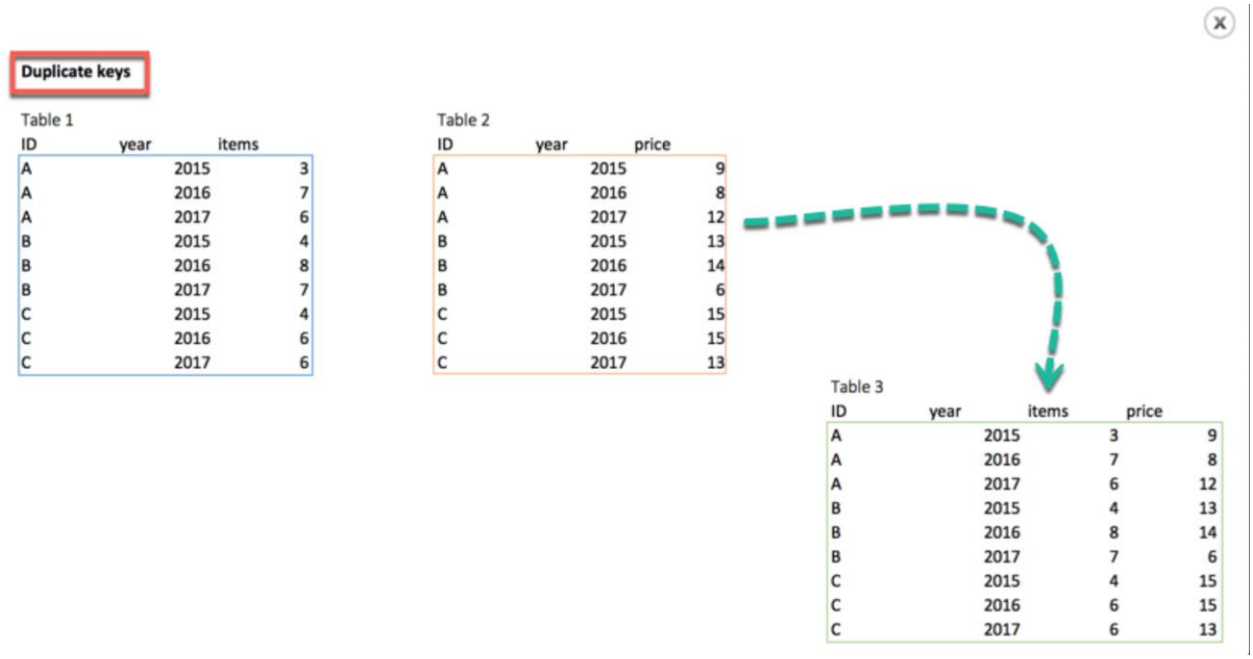
basically left_join treats the left or first variable mentioned as the base, that’s why F is included and E is excluded

right_join treats the right or second variable as base, so that case E would included and F excluded

inner_join excludes all unmatched observations, meaning both E and F would be excluded

full_join keeps all observations

Multiple key pairs



R won’t know how to match ID since there are two A with different years. Here, have to specify two or more key variables.

```
left_join(df_primary, df_secondary, by = c('ID', 'year'))
```

Data Cleaning functions

- gather(): Transform the data from wide to long
- spread(): Transform the data from long to wide
- separate(): Split one variable into two
- unit(): Unit two variables into one

Gather

```
gather(data, key, value, na.rm = FALSE)
Arguments:
        -data: The data frame used to reshape the dataset
        -key: Name of the new column created
        -value: Select the columns used to fill the key column
        -na.rm: Remove missing values. FALSE by default
```

The tutorial did it in a complicated way using %>% which is sort of like "then"

- x %>% impute %>% shuffle %>% pivot
- ***is the same as***

- pivot(shuffle(impute(x)))
- ***lets you input commands on inner parts first, good for longer and more complicated codes***

Tutorials way

```
> tidier <-messy %>%
+   gather(quarter, growth, q1_2017:q4_2017)
```

My way

```
>
> a = gather(messy, quarter, growth, q1_2017:q4_2017)
> a
   country quarter growth
1        A q1_2017   0.03
2        B q1_2017   0.05
3        C q1_2017   0.01
4        A q2_2017   0.05
5        B q2_2017   0.07
```

```
6       C q2_2017   0.02
7       A q3_2017   0.04
8       B q3_2017   0.05
9       C q3_2017   0.01
10       A q4_2017   0.03
11       B q4_2017   0.02
12       C q4_2017   0.04
```

It's the same thing

```
> a==tidier
      country quarter growth
 [1,]    TRUE    TRUE   TRUE
 [2,]    TRUE    TRUE   TRUE
 [3,]    TRUE    TRUE   TRUE
 [4,]    TRUE    TRUE   TRUE
 [5,]    TRUE    TRUE   TRUE
 [6,]    TRUE    TRUE   TRUE
 [7,]    TRUE    TRUE   TRUE
 [8,]    TRUE    TRUE   TRUE
 [9,]    TRUE    TRUE   TRUE
[10,]    TRUE    TRUE   TRUE
[11,]    TRUE    TRUE   TRUE
[12,]    TRUE    TRUE   TRUE
```

Spread () – same idea as gather, but makes long into wide

Separate () – split one column into two. Useful for dates.

```
separate(data, col, into, sep= "", remove = TRUE)
arguments:
        -data: The data frame used to reshape the dataset
        -col: The column to split
        -into: The name of the new variables
        -sep: Indicates the symbol used that separates the variable, i.e.:  "-", "_", "&"
        -remove: Remove the old column. By default sets to TRUE.
```

Syntax:

```
separate(data, col, into, sep= "", remove = TRUE)
arguments:
        -data: The data frame used to reshape the dataset
        -col: The column to split
```

-into: The name of the new variables
-sep: Indicates the symbol used that separates the variable, i.e.: "-", "_", "&"
-remove: Remove the old column. By default sets to TRUE.

We can split the quarter from the year in the tidier dataset by applying the separate() function.

```
separate_tidier <-tidier %>%
separate(quarter, c("Qrt", "year"), sep ="_")
head(separate_tidier)
```

The above commands are the same as…

```
> separate (tidier, quarter, c("Qrt", "year"), sep ="_")
   country Qrt year growth
1        A  q1 2017   0.03
2        B  q1 2017   0.05
3        C  q1 2017   0.01
4        A  q2 2017   0.05
5        B  q2 2017   0.07
6        C  q2 2017   0.02
7        A  q3 2017   0.04
8        B  q3 2017   0.05
9        C  q3 2017   0.01
10       A  q4 2017   0.03
11       B  q4 2017   0.02
12       C  q4 2017   0.04
```

Because we wanted to separate q1-4 and the years (i.e. q1_2017) we have to specify sep="_" because "_" is what's separating the variables

unite()

The unite() function concanates two columns into one.

Syntax:

```
unit(data, col, conc ,sep= "", remove = TRUE)
arguments:
```

-data: The data frame used to reshape the dataset
-col: Name of the new column
-conc: Name of the columns to concatenate
-sep: Indicates the symbol used that unites the variable, i.e: "-", "_", "&"
-remove: Remove the old columns. By default, sets to TRUE

```
unit_tidier <- separate_tidier %>%
```

```
  unite(Quarter, Qrt, year, sep ="_")
head(unit_tidier)
```

The above is the same as below.

```
a = unite(separate_tidier, Quarter, Qrt, year, sep = "_")
> a
   country Quarter growth
1        A q1_2017   0.03
2        B q1_2017   0.05
3        C q1_2017   0.01
4        A q2_2017   0.05
5        B q2_2017   0.07
6        C q2_2017   0.02
7        A q3_2017   0.04
8        B q3_2017   0.05
9        C q3_2017   0.01
10       A q4_2017   0.03
11       B q4_2017   0.02
12       C q4_2017   0.04
```

[Merge Data Frames in R: Full and Partial Match]

Full Match

Used when two data frames have one category with all the same values

Merge

```
merge(x, y, by.x = x, by.y = y)

Arguments:

-x: The origin data frame

-y: The data frame to merge

-by.x: The column used for merging in x data frame. Column x to merge on

-by.y: The column used for merging in y data frame. Column y to merge on
```

```
> producers <- data.frame(
+     surname =  c("Spielberg","Scorsese","Hitchcock","Tarantino","Polanski"),
+     nationality = c("US","US","UK","US","Poland"),
+     stringsAsFactors=FALSE)
> movies <- data.frame(
+     surname = c("Spielberg",
+                 "Scorsese",
+                 "Hitchcock",
+                 "Hitchcock",
+                 "Spielberg",
+                 "Tarantino",
+                 "Polanski"),
+     title = c("Super 8",
+               "Taxi Driver",
+               "Psycho",
+               "North by Northwest",
+               "Catch Me If You Can",
+               "Reservoir Dogs","Chinatown"),
+     stringsAsFactors=FALSE)
```

We add stringsAsFactors=FALSE in the data frame because we don't want R to convert string as factor, we want the variable to be treated as character.

```
# Merge two datasets
m1 <- merge(producers, movies, by.x = "surname")
```

```
m1
```

Output:

```
surname         nationality         title
1 Hitchcock             UK          Psycho
2 Hitchcock             UK          North by Northwest
3 Polanski              Poland      Chinatown
4 Scorsese              US          Taxi Driver
5 Spielberg             US          Super 8
6 Spielberg             US          Catch Me If You Can
7 Tarantino             US          Reservoir Dogs
```

#If we enter this code, the variable that represent surname is different for x and y data frame.

```
colnames(movies)[colnames(movies) == 'surname'] <- 'name'
```

How do you combine?

```
m2 <- merge(producers, movies, by.x = "surname", by.y = "name")

##surname    nationality         title
## 1 Hitchcock      UK           Psycho
## 2 Hitchcock      UK           North by Northwest
## 3 Polanski       Poland       Chinatown
## 4 Scorsese       US           Taxi Driver
## 5 Spielberg      US           Super 8
## 6 Spielberg      US           Catch Me If You Can
```

<u>Partial Match</u>

Used when two data frames don't have a category with all the same values.

For example if we add a new producer "Lucas" to "producers" but not "movies" then how do we add?

```
add_producer <-  c('Lucas', 'US')
producers <- rbind(producers, add_producer)
```

- ***Observation Lucas was added to data frame "producers." Here Lucas has all the same variables as "movies," but is missing "title."***

```
> merge (producers, movies, by.x = "surname", by.y = "name", all.x=TRUE)
   surname nationality              title
1 Hitchcock          UK             Psycho
2 Hitchcock          UK  North by Northwest
3     Lucas          US               <NA>
4  Polanski      Poland           Chinatown
5  Scorsese          US        Taxi Driver
6 Spielberg          US            Super 8
7 Spielberg          US Catch Me If You Can
8 Tarantino          US      Reservoir Dogs
```

This specified the different names used (by.x = “”, and by.y = “”).

All.x= TRUE → makes it so that all variables, even unmatched ones are displayed.

[Functions in R Programming (with Example)]

Function in R

Syntax

```
function (arglist)  {
 #Function body
}
```

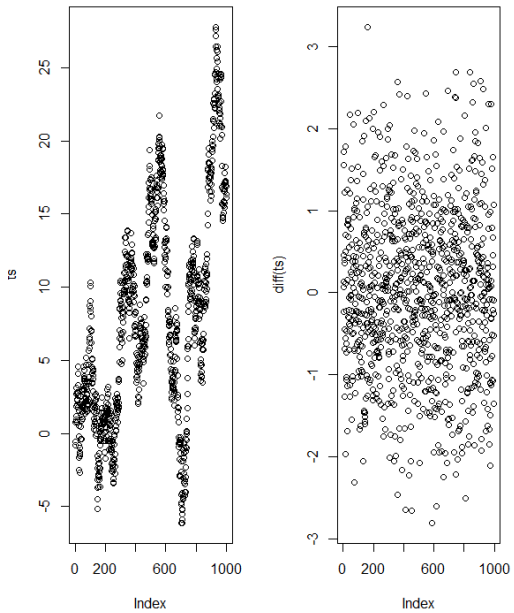
General Functions

Diff() function

- Used for time series data, since you need lag values. This process is called stationary process (**allows constant mean, variance and autocorrelation over time)**

```
> set.seed(123)
> ## Create the data
> x = rnorm(1000)
> ts <- cumsum(x)

> plot (ts)
> plot(diff(ts))
```



Not sure what this all means. Definitions of Lag Values and Stationary Process is provided

Lag Values

In statistics and econometrics, a distributed lag **model is a model for time series data in which a regression equation is used to predict current** values **of a dependent variable based on both the current** values **of an explanatory variable and the** lagged**(past period)** values **of this explanatory variable.**

Stationary Process

A stationary **(**time**)** series **is one whose statistical properties such as the mean, variance and autocorrelation are all constant over** time**. Hence, a** non-stationary series **is one whose statistical properties change over** time**.**

Why do we use stationary time series?
A common assumption in many **time series** techniques **is** that the data **are stationary**.
A **stationary** process has the property that the mean, variance and autocorrelation structure **do** not change over **time**. ... If the data contain a trend, **we can** fit some type of curve to the data and then model the residuals from that fit.

Length() function

This gives you the number of columns the data frame or matrix has

```
length(dt)
```

this gives you the number of rows

```
length(dt[,1])
```

Math Functions

Operator	Description
abs(x)	Takes the absolute value of x
log(x,base=y)	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm

exp(x)	Returns the exponential of x
sqrt(x)	Returns the square root of x
factorial(x)	Returns the factorial of x (x!)

All of these functions can be applied to single digits, vectors, matrix data frames, etc…

For example…

```
> vec = c(1,2,3,4)
> exp (vec)
[1]  2.718282  7.389056
[3] 20.085537 54.598150


> sqrt(matrix(1:4, nrow = 4, ncol = 4))
         [,1]     [,2]     [,3]
[1,] 1.000000 1.000000 1.000000
[2,] 1.414214 1.414214 1.414214
[3,] 1.732051 1.732051 1.732051
[4,] 2.000000 2.000000 2.000000
         [,4]
[1,] 1.000000
[2,] 1.414214
[3,] 1.732051
[4,] 2.000000
```

Statistical Functiona

Basic statistic functions

Operator	Description
mean(x)	Mean of x
median(x)	Median of x

var(x)	Variance of x
sd(x)	Standard deviation of x
scale(x)	Standard scores (z-scores) of x
quantile(x)	The quartiles of x
summary(x)	Summary of x: mean, min, max etc..

Self-explanatory. Similar to STATA

Side notes

- ***when you create variable like this… a=2 → a is automatically considered a factor***
- ***but if you create it like "a" = 2 → then it is a string variable***

Creating a Function

```
> square_function<- function(n)
+ {
+     n^2
+ }
> square_function(2)
[1] 4
```

Can also be written like this…

```
> add_2 = function (x) {x+2}
> add_2 (2)
[1] 4
```

Use this to erase/remove function you created

- ***rm (add_2)***

Environment Scoping

In R, the environment **is a** collection **of objects like functions, variables, data frame, etc.**

- **Highest level environment = global environment a.k.a R_GlobalEnv**
- **Second is Local Environment**

```
> ls(environment())
 [1] "a"
 [2] "add_2"
 [3] "add_producer"
 [4] "diff_ts"
 [5] "dt"
 [6] "mat"
 [7] "messy"
 [8] "movies"
 [9] "producers"
[10] "separate_tidier"
[11] "square_function"
[12] "tidier"
[13] "ts"
[14] "vec"
[15] "x"
```

Things assigned values in a function are stored locally, like the below function

```
add_2 = function (x)
+ {y=10
+  x+y}

> y
Error: object 'y' not found
```

To store the y value "globally" you can specify "y" outside of the function

Multi Arguments Function

```
times <- function(x,y) {
  x*y
            }
times(2,4)
```

This lets us specify both the x and y value each time

Exercise. How would we create a function for this equation?

$$normalize = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Where the X is variable representing a collection of observations. So basically, we want to apply this equation to each observation.

```
> x1 = rnorm(50, 5, 1.5)
> x2 = rnorm(50, 5, 1.5)
> df = data.frame (x1, x2)

> df$x1_norm = ((df$x1-min(df$x1))/(max(df$x1)-min(df$x1))
+ )
> df$x1_norm

[1] 0.199960521 0.339368912
[3] 0.177949719 0.592912635
[5] 0.726773455 1.000000000
[7] 0.522098656 0.185362507
[9] 0.700754897 0.668268128
[11] 0.358095439 0.000000000
[13] 0.629341036 0.729158585
[15] 0.342665839 0.568265623
[17] 0.347880764 0.607904623
[19] 0.390919248 0.329214791
[21] 0.453037826 0.345841496
```

Then you can go ahead and do this for x2

BUT DOING THIS IS PRONE TO MISTAKES, SO IT'S BETTER TO SEPARATE DIFFERENT FUNCTIONS AND COMBINE THEM

```
nominator <- x-min(x)
denominator <- max(x)-min(x)
normalize <- nominator/denominator
```

#We need to write return() so that when we specify normalize(), it'll give us the output we want.

Functions with Condition

split_data <- function(df, train = TRUE)

Arguments:
-df: Define the dataset
-train: Specify if the function returns the train set or test set. By default, set to TRUE

```
> length<- nrow(airquality)
> total_row <- length*0.8
> split <- 1:total_row
> split
[1]   1   2   3   4   5   6
  [7]   7   8   9  10  11  12
 [13]  13  14  15  16  17  18
 [19]  19  20  21  22  23  24
 [25]  25  26  27  28  29  30
 [31]  31  32  33  34  35  36
 [37]  37  38  39  40  41  42
 [43]  43  44  45  46  47  48
 [49]  49  50  51  52  53  54
 [55]  55  56  57  58  59  60
```

And so on…until 122

```
> train_df <- airquality[split, ]
> train_df
```

	Ozone	Solar.R	Wind	Temp
1	41	190	7.4	67
2	36	118	8.0	72
3	12	149	12.6	74
4	18	313	11.5	62
5	NA	NA	14.3	56
6	28	NA	14.9	66
7	23	299	8.6	65

And so on until 122nd row…

So it's selecting the first 122 rows of the data

If we want to select the rows after 122 we can use this function (put – in front of split)

```
test_df <- airquality[-split, ]
```

You can do things like this

```
if (train ==TRUE){
    train_df <- airquality[split, ]
      return(train)
  } else {
    test_df <- airquality[-split, ]
      return(test)
  }
```

But the above only applies to "air quality" so to make it applicable to all data frames use this

```
split_data <- function(df, train = TRUE){
  length<- nrow(df)
  total_row <- length *0.8
  split <- 1:total_row
  if (train ==TRUE){
    train_df <- df[split, ]
      return(train_df)
  } else {
    test_df <- df[-split, ]
      return(test_df)
  }
}
```

As stated before "return" specifies what should be printed

Don't really understand the difference between train and test. Don't think I'll be needing it yet?

[IF, ELSE, ELSE IF]

Summary: use IF to start off and use ELSE if there are more conditions following it, otherwise use ELSE IF to end

```
> if (quantity<=3) {
+     print ("ok")
+ } else if (quantity <=5) {
+     print ("no")
+ } else if (quantity<=7) {
+     print ("yes")
+ } else {
+     print ("maybe")
+ }
[1] "ok"
Warning message:
In if (quantity <= 3) { :
  the condition has length > 1 and only the first element will be used
```

This command didn't work because this doesn't apply to vectors. If I wanted it to work, "quantity" would have to be a single value.

```
> quantity = 5
> if (quantity<=3) {
+     print ("ok")
+ } else if (quantity <=5) {
+     print ("no")
+ } else if (quantity<=7) {
+     print ("yes")
+ } else {
+     print ("maybe")
+ }
[1] "no"
```

Example 2:

VAT has different rate according to the product purchased. Imagine we have three different kind of products with different VAT applied:

Categories	Products	VAT
A	Book, magazine, newspaper, etc..	8%

B	Vegetable, meat, beverage, etc..	10%
C	Tee-shirt, jean, pant, etc..	20%

We can write a chain to apply the correct VAT rate to the product a customer bought.

```
category <- 'A'
price <- 10
if (category =='A'){
  cat('A vat rate of 8% is applied.','The total price is',price *1.08)
} else if (category =='B'){
    cat('A vat rate of 10% is applied.','The total price is',price *1.10)
} else {
    cat('A vat rate of 20% is applied.','The total price is',price *1.20)
}
```

Output:

```
# A vat rate of 8% is applied. The total price is 10.8
```

Tutorial doesn't explain what cat() does, but assuming it let's you print a statement and do calculations

Cat()

Concatenate And Print

Outputs the objects, concatenating the representations. `cat` performs much less conversion than `print`.

Keywords file, print, connection

Usage

```
cat(… , file = "", sep = " ", fill = FALSE, labels = NULL,
    append = FALSE)
```

Arguments

...	R objects (see 'Details' for the types of objects allowed).
file	A connection, or a character string naming the file to print to. If `""` (the default), `cat` prints to the standard output connection, the console unless redirected by `sink`. If it is `"|cmd"`, the output is piped to the command given by `cmd`, by opening a pipe connection.
sep	a character vector of strings to append after each element.
fill	a logical or (positive) numeric controlling how the output is broken into successive lines. If `FALSE` (default), only newlines created explicitly by `"\n"` are printed. Otherwise, the output is broken into lines with print width equal to the option `width` if `fill` is `TRUE`, or the value of `fill` if this is numeric. Non-positive `fill` values are ignored, with a warning.
labels	character vector of labels for the lines printed. Ignored if `fill` is `FALSE`.
append	logical. Only used if the argument `file` is the name of file (and not a connection or `"|cmd"`). If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`.

[For Loop for List and Matrix]

Syntax

```
For (i in vector) {
   Exp
}
```

R is going to do the "Exp" that is specified for all the i's in the vector

```
> a = c(1:4)
> a
[1] 1 2 3 4
> for (i in a) {
+     a[[i]]=i*3
+ }
> print (a)
[1]  3  6  9 12
```

Here, I created a vector and asked R to multiply 3 to each observation in a.

These set of commands makes you print the values of the list

```
fruit <- list(Basket = c('Apple', 'Orange', 'Passion fruit', 'Banana'),
Money = c(10, 12, 15), purchase = FALSE)
for (p  in fruit)
{
            print(p)
}
```

```
> mat <- matrix(1:10,nrow = 5, ncol =2)

> for (r in 1:nrow (mat))
+     for (c in 1:ncol(mat))
+         print(paste("Row", r, "and column", c, "have values of", mat[r, c]))
[1] "Row 1 and column 1 have values of 1"
[1] "Row 1 and column 2 have values of 6"
[1] "Row 2 and column 1 have values of 2"
[1] "Row 2 and column 2 have values of 7"
[1] "Row 3 and column 1 have values of 3"
[1] "Row 3 and column 2 have values of 8"
[1] "Row 4 and column 1 have values of 4"
[1] "Row 4 and column 2 have values of 9"
[1] "Row 5 and column 1 have values of 5"
[1] "Row 5 and column 2 have values of 10"
```

Have to use "for"
R represents the row number (same with c)
1:nrow is used to say "from the 1st row to the last row (or the total number of rows in mat)
Paste(), Doesn't explain but assuming it lets you specify statement along with values from variable name or in this case r and c
Paste()

Concatenate Strings

Concatenate vectors after converting to character.

Keywords character

Usage

```
paste (…, sep = " ", collapse = NULL)
paste0(…, collapse = NULL)
```

Arguments

...	one or more R objects, to be converted to character vectors.
sep	a character string to separate the terms. Not `NA_character_` .
collapse	an optional character string to separate the results. Not `NA_character_` .

[While Loop]

Syntax

```
while (condition) {
   Exp
}
```

```
begin = 1
> while (begin <= 10){
+     cat("Value is", begin)
+     begin = begin + 1
+     print (begin)
+ }
Value is 1[1] 2
Value is 2[1] 3
Value is 3[1] 4
Value is 4[1] 5
Value is 5[1] 6
Value is 6[1] 7
Value is 7[1] 8
Value is 8[1] 9
Value is 9[1] 10
Value is 10[1] 11
```

Scenario.
We bought stocks for 50 but want to short it if it goes below 45. We want to see how many loops it takes for the value of our stock to be less than 45.

sample(x, size,blah blah)
in our sample () function, we're choosing 1 random value (size) from -10 to 10

we don't include cat() in the function since we only want one value or else it'll do it for all price values greater than 45 and less than 50. Or else we would get this…

```
it took 2 loop before we short the price. The lowest price is 57[1] 2
it took 3 loop before we short the price. The lowest price is 47[1] 3
it took 4 loop before we short the price. The lowest price is 48[1] 4
it took 5 loop before we short the price. The lowest price is 43[1] 5
```

Sort of a nonsensical example, but it shows the logic of loops.

```
> while (price > 45){
+     price = stock + sample (-10:10,1)
+     loop = loop +1
```

```
+     print(loop)
+ }

> cat('it took',loop,'loop before we short the price. The lowest price is',price)

it took 5 loop before we short the price. The lowest price is 43
```

[apply(), lapply(), sapply(), tapply()]

Apply()

Syntax

apply(X, MARGIN, FUN)

-x: an array or matrix
-MARGIN: take a value or range between 1 and 2 to define where to apply the function:
-MARGIN=1`: the manipulation is performed on rows
-MARGIN=2`: the manipulation is performed on columns
-MARGIN=c(1,2)` the manipulation is performed on rows and columns
-FUN: tells which function to apply. Built functions like mean, median, sum, min, max and even user-defined functions can be applied>

It adds a function that you specify.

Pretty self-explanatory, but keep in mind that "X" needs a row and a column. So it wouldn't work for vectors.

```
> vec_app = apply(vec, MARGIN = 2, sum)
Error in apply(vec, MARGIN = 2, sum) : dim(X) must have a positive length
```

Lapply()

Syntax

lapply(X, FUN)

Arguments:
-X: A vector or an object
-FUN: Function applied to each element of x

Same as apply, but it's for lists and matrix. You do not need to specify a margin for this command

```
movies <- c("SPYDERMAN","BATMAN","VERTIGO","CHINATOWN")
movies_lower <-lapply(movies, tolower)
movies_lower <-unlist(lapply(movies,tolower))
str(movies_lower)
##  chr [1:4] "spyderman" "batman" "vertigo" "chinatown"
```

Sapply()

sapply(X, FUN)
Arguments:
-X: A vector or an object
-FUN: Function applied to each element of x

```
dt <- cars
lmn_cars <- lapply(dt, min)
smn_cars <- sapply(dt, min)
lmn_cars

## $speed
## [1] 4
## $dist
## [1] 2
```

Same as above, but it's for vectors

Slice Vector with Lapply or Sapply

The commands below created a function to return values of x that are greater than the average

```
below_ave <- function(x) {
   ave <- mean(x)
   return(x[x > ave])
}
dt_s<- sapply(dt, below_ave)
dt_l<- lapply(dt, below_ave)
identical(dt_s, dt_l)
```

identical() was used to see that both lapply and sapply returned the same output

The command return(x[x>ave]) → the part in [] is the parameters you specify. It's sort of like an IF

Tapply ()

Syntax

```
tapply(X, INDEX, FUN = NULL)
```

Arguments:
-X: An object, usually a vector
-INDEX: A list containing factor
-FUN: Function applied to each element of x

```
> head (iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
> tapply(iris$Sepal.Width, iris$Species, mean)
    setosa versicolor  virginica
     3.428      2.770      2.974
```

Basically, x is the vectors you want to change, and the index is the category (name) you want to use to identify the values with.

[Import Data into R: Read CSV, Excel, SPSS, Stata, SAS Files]

Read CSV

Syntax

read.csv(file, header = TRUE, sep = ",")

Argument:

- **file**: PATH where the file is stored
- **header**: confirm if the file has a header or not, by default, the header is set to TRUE
- **sep**: the symbol used to split the variable. By default, `,`.

The below is the PATH that you should specify in "file."

"C:\Users\USERNAME\Downloads\FILENAME.csv"

Read Excel files

require(readxl)

This command is for opening excel files

Read_excel()

Syntax

read_excel(PATH, sheet = NULL, range= NULL, col_names = TRUE)

arguments:

-PATH: Path where the excel is located
-sheet: Select the sheet to import. By default, all
-range: Select the range to import. By default, all non-null cells
-col_names: Select the columns to import. By default, all non-null columns

This command allows you to specify specific parts of the excel sheet you want to open

For the range, you can specify the cell rows with cell_rows(x:y). The variable name for the rows in excel are saved as "cell_rows"

Excel_sheets ()

```
example <- readxl_example("datasets.xlsx")
excel_sheets(example)
```

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

With these set of commands, you can see which sheets are available in the excel file

```
iris <-read_excel(example, n_max =5, col_names =TRUE)
```

```
> iris
# A tibble: 5 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
>
```

This gives you just the first 5 "n" with the headers.
If you put col_names = FALSE, R will output the data without the column headers

Import Data from other Statistical Software

- SAS: read_sas()
- STATA: read_dta() (or read_stata(), which are identical)
- SPSS: read_sav() or read_por(). We need to check the extension

For all of these you can simple assign a variable to the PATH and open it as such.

```
SAS_Data = 'https://stats.idre.ucla.edu/wp-content/uploads/2016/02/binary.sas7bdat'
```