A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

Create Data Frame

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Print the data frame.
print(emp.data)
```

When we execute the above code, it produces the following result −

```
 emp_id    emp_name     salary     start_date
1     1     Rick        623.30     2012-01-01
2     2     Dan         515.20     2013-09-23
3     3     Michelle    611.00     2014-11-15
4     4     Ryan        729.00     2014-05-11
5     5     Gary        843.25     2015-03-27
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using **str()** function.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Get the structure of the data frame.
str(emp.data)
```

When we execute the above code, it produces the following result −

```
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
 $ start_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...
```

Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying **summary()** function.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Print the summary.
print(summary(emp.data))
```

When we execute the above code, it produces the following result −

```
     emp_id    emp_name             salary        start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                      3rd Qu.:729.0   3rd Qu.:2014-11-15
 Max.   :5                      Max.   :843.2   Max.   :2015-03-27
```

Extract Data from Data Frame

Extract specific column from a data frame using column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Extract Specific columns.
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

When we execute the above code, it produces the following result −

```
  emp.data.emp_name emp.data.salary
1              Rick          623.30
2               Dan          515.20
3          Michelle          611.00
4              Ryan          729.00
5              Gary          843.25
```

Extract the first two rows and then all columns

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Extract first two rows.
result <- emp.data[1:2,]
print(result)
```

When we execute the above code, it produces the following result −

```
  emp_id    emp_name   salary    start_date
1      1     Rick      623.3     2012-01-01
2      2     Dan       515.2     2013-09-23
```

Extract 3rd and 5th row with 2nd and 4th column

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

                start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)

# Extract 3rd and 5th row with 2nd and 4th column.
result <- emp.data[c(3,5),c(2,4)]
print(result)
```

When we execute the above code, it produces the following result −

```
  emp_name start_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),
   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)

# Add the "dept" coulmn.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

When we execute the above code, it produces the following result −

```
  emp_id   emp_name    salary    start_date       dept
1     1    Rick        623.30    2012-01-01       IT
2     2    Dan         515.20    2013-09-23       Operations
3     3    Michelle    611.00    2014-11-15       IT
4     4    Ryan        729.00    2014-05-11       HR
5     5    Gary        843.25    2015-03-27       Finance
```

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   dept = c("IT","Operations","IT","HR","Finance"),
   stringsAsFactors = FALSE
)

# Create the second data frame
emp.newdata <-      data.frame(
   emp_id = c (6:8),
   emp_name = c("Rasmi","Pranab","Tusar"),
   salary = c(578.0,722.5,632.8),
   start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),
   dept = c("IT","Operations","Fianance"),
   stringsAsFactors = FALSE
)
```

```
# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

When we execute the above code, it produces the following result −

```
  emp_id    emp_name    salary    start_date      dept
1      1    Rick        623.30    2012-01-01      IT
2      2    Dan         515.20    2013-09-23      Operations
3      3    Michelle    611.00    2014-11-15      IT
4      4    Ryan        729.00    2014-05-11      HR
5      5    Gary        843.25    2015-03-27      Finance
6      6    Rasmi       578.00    2013-05-21      IT
7      7    Pranab      722.50    2013-07-30      Operations
8      8    Tusar       632.80    2014-06-17      Fianance
```

The dim Function in R

Basic R Syntax:

```
dim(data)
```

The dim function of the R programming language **returns the dimension** (e.g. the number of columns and rows) of a matrix, array or data frame.

Example 1: Dimension of Matrix or Data Frame

Let's first create some example data, before we start with the application of dim in R:

```
set.seed(62626)                             # Set Seed for reproducibility
N <- 500                                    # Sample size

x1 <- round(rnorm(N, 0, 10))                # Create 5 random variables
x2 <- round(runif(N, 5, 10))
x3 <- round(runif(N, 1, 3), 1)
x4 <- round(runif(N, 10, 20))
x5 <- rpois(N, 5)

data <- data.frame(x1, x2, x3, x4, x5)    # Data frame with 5 columns
head(data)                                  # First 6 rows of data.frame
```

X1	X2	X3	X4	X5
-7	7	1.7	11	3
-16	8	2.4	18	7
-7	5	2	17	4
12	7	2.1	14	3
5	9	2.7	14	7
19	8	2	20	5

Table 1: First 6 Rows of Our Example Data Frame for the Application of dim in R.

Table 1 illustrates how our example data.frame looks like. It's easy to see that the data consists of 5 columns. But how many rows? Let's check with the dim R function:

```
dim(data)                                 # Apply dim function to data.frame
# 500 5
```

After applying the dim function in R (I use the RStudio interface), we get two numbers back. The first number reflects the number of rows; and the second number reflects the number of columns.

In other words: Our data frame consists of 500 rows and 5 columns.

The same procedure could be applied to a matrix. Let's convert our data to the matrix format and check if it works:

```
data_matrix <- as.matrix(data)            # Convert data.frame to matrix
dim(data_matrix)                          # Apply dim function to matrix
# 500 5
```

Same result as before – perfect!

Example 2: dim of List in R

Sometimes, it is useful to use dim for a list object in R. That task is easily done with a combination of dim() and sapply().

First, let's create a list in R:

```
data_list <- list()                         # Create empty list object
data_list[[1]] <- data                      # First entry of list
data_list[[2]] <- data[1:10, ]              # Second entry (subset of data)
data_list[[3]] <- data[5:67, c(1, 3, 5)] # Third entry (other subset of data)
```

Now, let's extract the dimensions of each list element:

```
sapply(data_list, dim)                      # Get dimension of all list entries
#       [,1] [,2] [,3]
# [1,]   500   10   63
# [2,]     5    5    3
```

The combination of dim and sapply returns a matrix to the RStudio console. Each column of this matrix reflects the dimension of one list element:

- List entry 1: 500 rows; 5 columns
- List entry 2: 10 rows; 5 columns
- List entry 3: 63 rows; 3 columns

Example 3: dim in R Returns NULL – What's the Problem?

A common mistake is the application of dim to a one dimensional vector or array. Let's see what happens, when we do this.

I'll first create an example vector…

```
vec1 <- c(5, 9, - 20, 3, 17, 18, 2)        # Example vector
```

…and then I'll apply the dim function:

```
dim(vec1)                                   # Apply dim function to vector
# NULL
```

As you see: That doesn't work!

In case you want to get the number of entries of a vector, you have to use the length function:

```
length(vec1)                              # Get length of vector or array
# 7
```

The nrow Function in R

Basic R Syntax:

```
nrow(data)
```

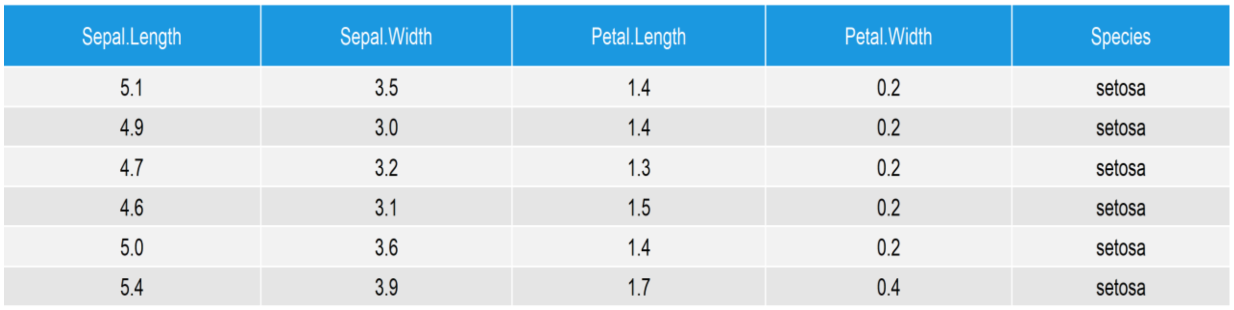
The nrow R function **returns the number of rows** that are present in a data frame or matrix. Above, you can find the R code for the usage of nrow in R.

You want to know more details? In this article, I'm going to provide you with several reproducible **examples of typical applications** of the nrow function in R.

Example 1: Count the Number of Rows of a Data Frame

For the following example, I'm going to use the iris data set. Load the data set:

```
data(iris)          # Load the iris data set
head(iris)          # Head of the iris data set
```



Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

Table 1: Iris Data Frame as Example for the Application of nrow in R.

After loading the data frame in R, we can apply the nrow function as follows:

```
nrow(iris)          # Number of rows
# 150
```

The number of lines of the iris database is 150.

Using nrow in R with Condition

Let's assume we want to count the rows of the iris data set where the variable Sepal.Length is larger than 5. With the following R code, we can examine this condition:

```
nrow(iris[iris$Sepal.Length > 5, ])            # Number of lines that meet the condition
# 118
```

118 rows (i.e. observations) have a Sepal.Length larger than 5.

Example 3: nrow with NA or NULL

Often, your data will have missing values (usually labeled as NA or NULL). You want to know, how many complete rows are available in your data? The complete.cases function can help:

```
iris_miss <- iris                              # Duplicate data
iris_miss$Species[c(3, 17, 55, 127)] <- NA     # Insert some NA values to Species
nrow(iris_miss[complete.cases(iris_miss), ])   # Count number of complete rows
# 146
```

For Loop Using nrow in R

In the next example, I'm showing you how to use the nrow R function as condition within a for loop – Probably the situation where I use nrow the most often.

Assume that we want to calculate the cumulative sum of the column Petal.Width:

```
cumsum_Petal.Width <- iris$Petal.Width[1]    # Create new vector object

for(i in 2:nrow(iris)) {                     # Use nrow as condition
  cumsum_Petal.Width[i] <-                   # Calculate cumulative sum
    cumsum_Petal.Width[i - 1] + iris$Petal.Width[i]
}

cumsum_Petal.Width                           # Print vector to RStudio console
# 0.2   0.4   0.6   0.8   1.0   1.4   1.7   1.9   2.1   2.2
```

Note: The cumulative sum could be calculated much easier with the cumsum R function. The for loop above is just for illustration.

The ncol Function in R

Basic R Syntax:

```
ncol(data)
```

The ncol R function **returns the number of columns** of a matrix or data frame. Above, you can find the command for the application of ncol in the R programming language.

You'd like to hear some more details? In the following tutorial, I'll provide you with several **examples of the usage** of the ncol function in R.

Count Number of Columns of a Data Frame

Before we can dive into the application of the ncol command in R, let's create an example data frame:

```
set.seed(99999)                        # Seed for reproducibility
N <- 100                               # Sample size
```

```
x1 <- round(runif(N, 1, 10))           # Column 1
x2 <- round(runif(N, 0, 3))            # Column 2
x3 <- round(runif(N, 1, 5))            # Column 3

data_frame <- data.frame(x1, x2, x3)  # Data frame with 3 columns
head(data_frame)                       # First 6 rows
```

X1	X2	X3
4	2	1
9	2	5
4	2	2
8	2	2
3	2	4
2	1	3

Table 1: Example Data for the Application of the ncol R function.

As you can see based on Table 1, our data frame consists of 3 columns. Let's check how we could investigate on that with the ncol function in R:

```
ncol(data_frame)                       # Count the number of columns
# 3
```

ncol returns the number 3 – seems correct!

Count the Number of Columns of a Matrix

The ncol function is easy to apply – also to matrices! Even if our data has the class matrix, we can apply the ncol command in the same manner.

First, let's convert the data frame we used before into a matrix:

```
mat <- as.matrix(data_frame)
```

Then, let's apply the ncol function:

```
ncol(mat)
# 3
```

Still 3 – very good.

ncol Returns NULL – A Common Mistake

A mistake that I see quite often is that people try to apply ncol to a vector (often, because they falsely think that their data is in data frame or matrix format).

The result is that R returns NULL, instead of the number of columns. Confusing…

I'll illustrate that with some R code:

```
set.seed(716253)                       # Set seed
vec1 <- rnorm(10, 5, 2)                # Some random data vector

ncol(vec1)                             # Apply the ncol R command
# NULL
```

As you can see, the ncol command is not working for vectors. If you want to know the amount of values of a vector, you have to use the transpose function…

```
ncol(t(vec1))                          # Transpose function in R
# 10
```

…or even easier: the length function.

```
length(vec1)                           # Length function in R
# 10
```

str() Function in R ,Compactly Display Structure of Object

Display Structure of Data Frame Using str() Function

Example 1 explains how to use the str function to print information on the structure of a data frame object.

First, we have to create an exemplifying data frame in R:

```
data <- data.frame(x1 = 1:5,          # Create example data frame
                   x2 = letters[1:5],
                   x3 = factor(c("yes", "no", "yes", "yes", "no")))
data                                  # Print example data frame
```

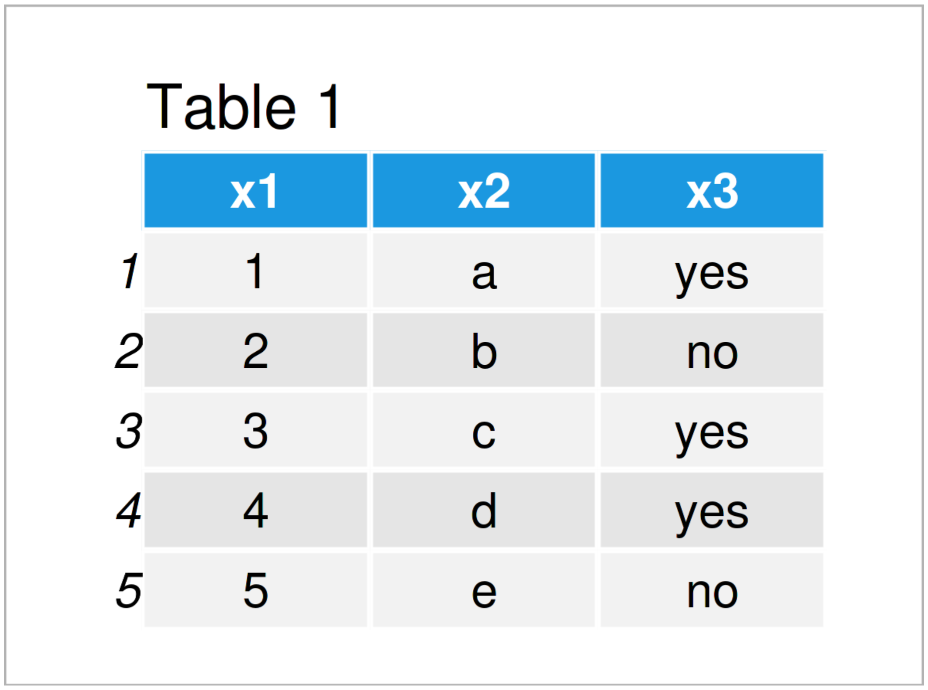


Table 1

	x1	x2	x3
1	1	a	yes
2	2	b	no
3	3	c	yes
4	4	d	yes
5	5	e	no

As shown in Table 1, we have created a data frame object with three columns by executing the previous R programming syntax.

Let’s apply the str function to this data frame:

```
str(data)                               # Apply str to data frame
# 'data.frame':5 obs. of  3 variables:
#  $ x1: int  1 2 3 4 5
#  $ x2: chr  "a" "b" "c" "d" ...
#  $ x3: Factor w/ 2 levels "no","yes": 2 1 2 2 1
```

Have a look at the previous output of the RStudio console. It shows some information on each of the variables of our data set.

The variable x1 is an integer and contains the values 1 to 5, the variable x2 is a character string and contains letters such as “a”, “b”, “c”, and “d”, and the variable x3 is a factor and contains the two factor levels “no” and “yes”.

Display Structure of List Using str() Function

In Example 2, I’ll illustrate how to check the structure of a list object using the str function.

Let’s create an example list:

```
my_list <- list(letters[3:1],           # Create example list
                555,
                c(1, 3, 5))
my_list                                 # Print example list
# [[1]]
# [1] "c" "b" "a"
#
# [[2]]
# [1] 555
#
# [[3]]
# [1] 1 3 5
```

Next, we can apply the str function to our list as shown below:

```
str(my_list)                            # Apply str to list
```

```
# List of 3
#  $ : chr [1:3] "c" "b" "a"
#  $ : num 555
#  $ : num [1:3] 1 3 5
```

The RStudio console output shows that our list has three list elements. The first list element is a character string containing the letters “c”, “b”, and “a”, the second list element is the numeric value 555, and the third list element is a numeric vector containing the values 1, 3, and 5.

Display Structure of Vector Using str() Function

The following syntax shows how to use the str command to evaluate the structure of a vector object.

For this, we first have to create a vector in R:

```
vec <- c(1, 5, 3, 7, 8, 7, 7, 3)     # Create example vector
vec                                  # Print example vector
# [1] 1 5 3 7 8 7 7 3
```

Next, we can apply the str function to this vector:

```
str(vec)                             # Apply str to vector
#  num [1:8] 1 5 3 7 8 7 7 3
```

The previous output illustrates that our vector is numeric and ranges from 1 to 8.

Definition & Basic R Syntax of summary Function

Definition: The summary R function computes summary statistics of data and model objects.

Basic R Syntax: Please find the basic R programming syntax of the summary function below.

```
summary(data)                       # Basic R syntax of summary function
```

In the following, I'll show three examples for the application of the summary function in R.

Applying summary Function to Vector

Example 1 illustrates how to apply the summary function to a numeric vector. First, we have to create a numeric vector in R:

```
vec <- 1:10                         # Create example vector
vec                                 # Print example vector
# 1   2   3   4   5   6   7   8   9 10
```

As you can see based on the previous RStudio console output, our example vector ranges from 1 to 10.

Now, we can use the summary command to calculate summary statistics of our vector:

```
summary(vec)                        # Apply summary function to vector
# Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
# 1.00    3.25    5.50    5.50    7.75   10.00
```

The summary function returned descriptive statistics such as the minimum, the first quantile, the median, the mean, the 3rd quantile, and the maximum value of our input data.

Applying summary Function to Data Frame

We can also apply the summary function to other objects. The following R programming syntax shows how to compute descriptive statistics of a data frame. First, we have to construct a data frame in R:

```
data <- data.frame(x1 = 1:5,     # Create example data frame
                   x2 = letters[1:5],
                   x3 = 3)
data                             # Print example data frame
#   x1 x2 x3
# 1  1  a  3
# 2  2  b  3
# 3  3  c  3
# 4  4  d  3
# 5  5  e  3
```

Our data frame contains five rows and three columns. We can now use the summary function to return summary statistics for each of the variables of this data frame to the RStudio console:

```
summary(data)                    # Apply summary function to data frame
#         x1     x2          x3
# Min.   :1   a:1   Min.   :3
# 1st Qu.:2   b:1   1st Qu.:3
# Median :3   c:1   Median :3
# Mean   :3   d:1   Mean   :3
# 3rd Qu.:4   e:1   3rd Qu.:3
# Max.   :5         Max.   :3
```

Change column name of a given DataFrame in R

A data frame is a tabular structure with fixed dimensions, of each rows as well as columns. It is a two-dimensional array like object with numerical, character based or factor-type data. Each element belonging to the data frame is indexed by a unique combination of the row and column number respectively. Column names are addressed by unique names.

Method 1: using colnames() method
colnames() method in R is used to rename and replace the column names of the data frame in R.

The columns of the data frame can be renamed by specifying the new column names as a vector. The new name replaces the corresponding old name of the column in the data frame. The length of new column vector should be equivalent to the number of columns originally. Changes are made to the original data frame.

Syntax:
colnames(df) <- c(new_col1_name,new_col2_name,new_col3_name)

Example:

- R

```
# declaring the columns of data frame

df = data.frame(

col1 = c('A', 'B', 'C', 'J', 'E', NA,'M'),

col2 = c(12.5, 9, 16.5, NA, 9, 20, 14.5),

col3 = c(NA, 3, 2, NA, 1, NA, 0))

# printing original data frame

print("Original data frame : ")

print(df)



print("Renaming columns names ")



# assigning new names to the columns of the data frame

colnames(df) <- c('C1','C2','C3')

# printing new data frame
```

```
print("New data frame : ")

print(df)
```

Output:

```
[1] "Original data frame : "
 col1 col2 col3
1    A 12.5   NA
2    B  9.0    3
3    C 16.5    2
4    J   NA   NA
5    E  9.0    1
6 <NA> 20.0   NA
7    M 14.5    0
[1] "Renaming columns names "
[1] "New data frame : "
   C1   C2 C3
1    A 12.5 NA
2    B  9.0  3
3    C 16.5  2
4    J   NA NA
5    E  9.0  1
6 <NA> 20.0 NA
7    M 14.5  0
```

1(A) .Specific columns of the data frame can also be renamed using the position index of the respective column.

Syntax:

colnames(df)[col_indx] <- "new_col_name_at_col_indx"

Approach

- Create dataframe
- Select the column to be renamed by index
- Provide a suitable name
- Change using colnames() function

Example:

- R

```
# declaring the columns of data frame

df = data.frame(

col1 = c('A', 'B', 'C', 'J', 'E', NA,'M'),

col2 = c(12.5, 9, 16.5, NA, 9, 20, 14.5),

col3 = c(NA, 3, 2, NA, 1, NA, 0))



# printing original data frame

print("Original data frame : ")

print(df)



print("Renaming columns names ")



# assigning the second column name to a new name

colnames(df)[2] <- "new_col2"
```

```
# printing new data frame

print("New data frame : ")

print(df)
```

Output:

```
[1] "Original data frame : "
 col1 col2 col3
1    A 12.5   NA
2    B  9.0    3
3    C 16.5    2
4    J   NA   NA
5    E  9.0    1
6 <NA> 20.0   NA
7    M 14.5    0
[1] "Renaming columns names "
[1] "New data frame : "
 col1 new_col2 col3
1    A     12.5   NA
2    B      9.0    3
3    C     16.5    2
4    J       NA   NA
5    E      9.0    1
6 <NA>     20.0   NA
7    M     14.5    0
```

1(B). Column names can also be replaced by using the which(names(df)) function, which searches for the column with the specified old name and then replaces it with the new specified name instance.

Syntax:

colnames(dataframe)[which(names(dataframe) == “oldColName”)] <- “ne wColName”

Approach

- Create data frame
- Select name of the columns to be changed
- Provide a suitable name
- Use the function

Example:

- R

```
# declaring the columns of data frame

df = data.frame(

col1 = c('A', 'B', 'C', NA,'M'),

col2 = c(12.5, 9, 16.5,  20, 14.5),

col3 = c(NA, 3, 2, NA, 0))



# printing original data frame

print("Original data frame : ")

print(df)



print("Renaming columns names ")
```

```
# assigning the second column name to a new name



colnames(df)[2] <- "new_col2"



# printing new data frame

print("After changing the data frame col2 name : ")

print(df)



# replacing first column name

colnames(df)[which(names(df) == "col1")] <- "new_col1"



# printing new data frame

print("After changing the data frame col1 name : ")

print(df)
```

Output

```
[1] "Original data frame : "
 col1 col2 col3
1    A 12.5   NA
2    B  9.0    3
3    C 16.5    2
4 <NA> 20.0   NA
5    M 14.5    0
```

```
[1] "Renaming columns names "
[1] "After changing the data frame col2 name : "
 col1 new_col2 col3
1    A     12.5   NA
2    B      9.0    3
3    C     16.5    2
4 <NA>     20.0   NA
5    M     14.5    0
[1] "After changing the data frame col1 name : "
 new_col1 new_col2 col3
1        A     12.5   NA
2        B      9.0    3
3        C     16.5    2
4     <NA>     20.0   NA
5        M     14.5    0
```

Method 2: using setNames() method
setNames() method in R can also be used to assign new names to the columns contained within a list, vector or tuple. The changes have to be saved back then to the original data frame, because they are not retained.

Syntax:
setnames(df, c(names of new columns))

Approach

- Create data frame
- Rename column using function
- Display modified data frame

Example:

- R

```
# declaring the columns of data frame

df = data.frame(

col1 = c('A', 'B', 'C', NA,'M'),

col2 = c(12.5, 9, 16.5,  20, 14.5),

col3 = c(NA, 3, 2, NA, 0))



# printing original data frame

print("Original data frame : ")

print(df)



# print("Renaming columns names ")

# renaming all the column names of data frame

df <- setNames(df,
c("changed_Col1","changed_Col2","changed_Col3"))



print("Renamed data frame : ")

print(df)
```

Output

```
[1] "Original data frame : "
  col1 col2 col3
1    A 12.5   NA
```

```
2    B  9.0    3
3    C 16.5    2
4 <NA> 20.0   NA
5    M 14.5    0
[1] "Renamed data frame : "
  changed_Col1 changed_Col2 changed_Col3
1            A         12.5           NA
2            B          9.0            3
3            C         16.5            2
4         <NA>         20.0           NA
5            M         14.5            0
```

The head() and tail() function in R

The **head() and tail() function in R** are often used to read the first and last n rows of a dataset.

You may be a working professional, a programmer, or a novice learner, but there are some times where you required to read large datasets and analyze them.

It is really hard to digest a huge dataset which have 20+ columns or even more and have thousands of rows.

This article will address the head() and tail() functions in R, which returns the first and last n rows respectively.

Syntax of the head() and tail() functions

Let's quickly see what the head() and tail() methods look like

Head(): Function which returns the first n rows of the dataset.

```
head(x,n=number)
```

Copy

Tail(): Function which returns the last n rows of the dataset.

```
tail(x,n=number)
```
Copy

Where,

x = input dataset / dataframe.

n = number of rows that the function should display.

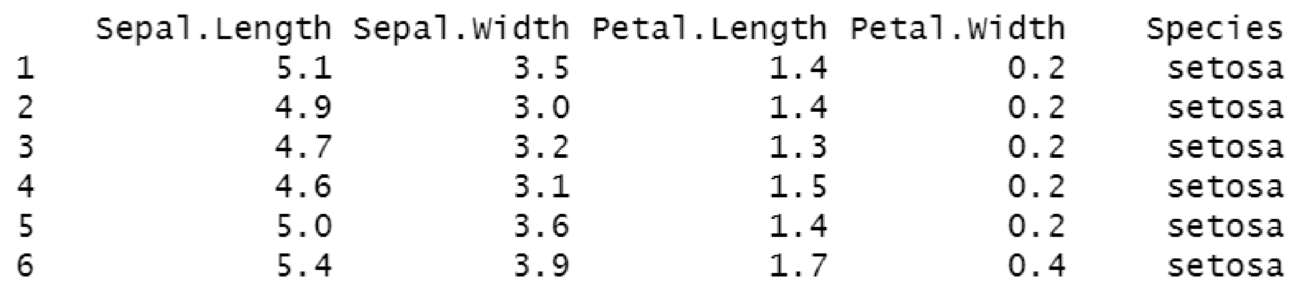
The head() function in R

The head() function in R is used to display the first *n* rows present in the input data frame.

In this section, we are going to get the first n rows using head() function.

For this process, we are going to import a dataset ‘iris’ which is available in R studio by default.

```
#importing the dataset
df<-datasets::iris

#returns first n rows of the data
head(df)
```
Copy



```
  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
```

You can see that the the head() function returned the first 6 rows present in the iris datatset.

The head() function with custom rows

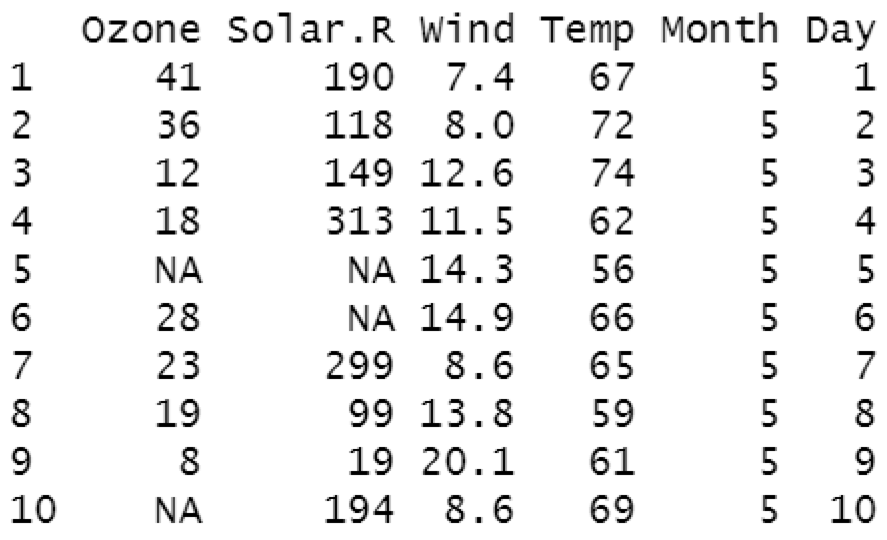
By default, the head() function returns the first 6 rows by default.

But what if, you want to see the first 10, 15 rows if a dataset?

Well, you may observed in the syntax that you can pass the number argument to the head function to display specific number of rows.

Let’s see how it works.

```
#importing the data
df<-datasets::airquality

#returns first 10 rows
head(df,n=10)
```
Copy



```
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67     5   1
2     36     118  8.0   72     5   2
3     12     149 12.6   74     5   3
4     18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
7     23     299  8.6   65     5   7
8     19      99 13.8   59     5   8
9      8      19 20.1   61     5   9
10    NA     194  8.6   69     5  10
```

You can now see the head() function returned the first 10 rows as specified by us in the input. You can also write the same query as head(df,10) and get the same results.

This is how head() function works.

head() function to get first n values in the specific column

Well, in the above sections, the head() function returned the whole set of values present in the first n rows of a dataset.

But do you know that the head() function in capable to returning the values of the particular column?

Yes, you read it right!

With a single piece of code, you can get the first n values of specified column.

```
#importing the data
df<-datasets::mtcars

#returns first 10 values in column 'mpg'
head(mtcars$mpg,10)
```
Copy

```
Output = 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
```
Copy

Just like the above sample, you can easily mention the required column name along with the required row count. That’s it.

The head() function will pierce into the data and returns the required.

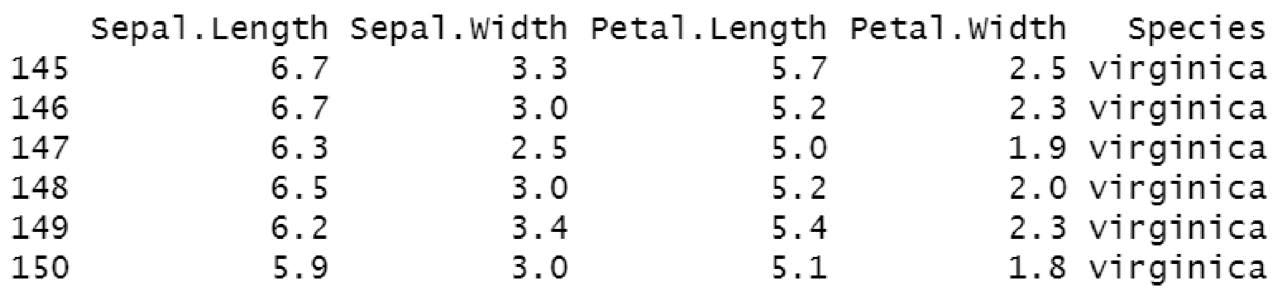
The tail() function in R

The tail() function in the R is particularly used to display the last n rows of the dataset, in contrary to the head() function.

This section will illustrate the tail() function and its usage in R.

For this purpose, we are using ‘airquality’ dataset.

```
#importing the dataset
df<-datasets::airquality

#returns last n rows of the data
tail(df)
```



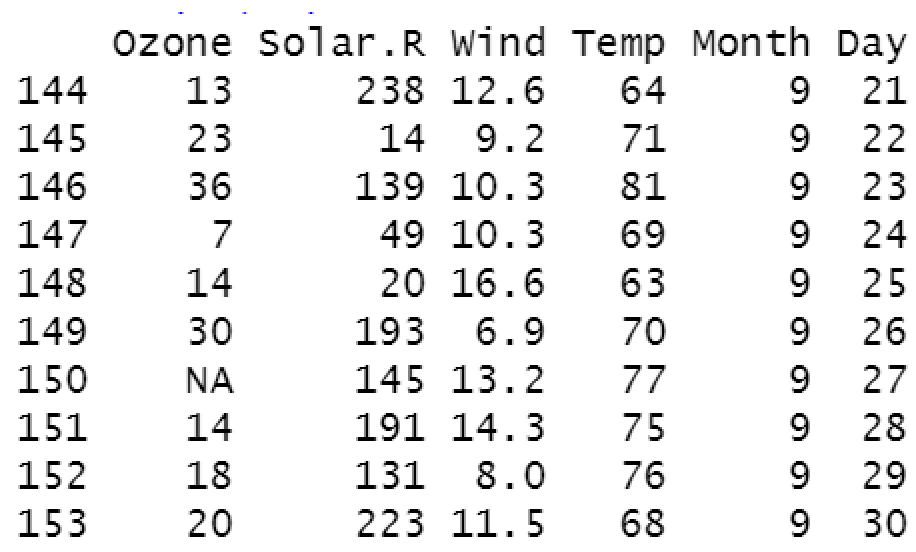
```
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

Well, in this output, you can see the last 6 rows of the iris dataset. This is what tail() function will do in R.

The tail() function with custom rows

Similar to the head() function, the tail() function can return the last n rows of the specified count.

```
#importing the data
df<-datasets::airquality

#returns the last 10 values
tail(df,10)
```



```
    Ozone Solar.R Wind Temp Month Day
144    13     238 12.6   64     9  21
145    23      14  9.2   71     9  22
146    36     139 10.3   81     9  23
147     7      49 10.3   69     9  24
148    14      20 16.6   63     9  25
149    30     193  6.9   70     9  26
150    NA     145 13.2   77     9  27
151    14     191 14.3   75     9  28
152    18     131  8.0   76     9  29
153    20     223 11.5   68     9  30
```

Here you can see, that the tail() function has returned the last 10 rows as specified by us in the code.

tail() function to get first n values in the specific column

The head() and tail() function does the same job in the quite opposite way.

You can use tail function to get last n values of a particular column as well.

Let’s see how it works!

```
#importing the data
df<-datasets::mtcars

#returns the last 10 values of column 'mpg'
tail(mtcars$mpg,10)
```

```
Output = 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

If you are able to get this output, congratulations! You have done it.

Just like this sample, you can specify the column name along with row count to get the required values.

edit: Invoke a Text Editor

Description :Invoke a text editor on an R object.

Usage

edit(name = NULL, file = "", title = NULL , editor = getOption("editor"), ...)

vi(name = NULL, file = "")

emacs(name = NULL, file = "")

pico(name = NULL, file = "")

xemacs(name = NULL, file = "")

xedit(name = NULL, file = "")

Arguments

name

a named object that you want to edit. If name is missing then the file specified by file is opened for editing.

file

a string naming the file to write the edited version to.

title

a display name for the object being edited.

editor

usually a character string naming (or giving the path to) the text editor you want to use. On Unix the default is set from the environment variables EDITOR or VISUAL if either is set, otherwise vi is used. On Windows it defaults to "internal", the script editor. On the macOS GUI the argument is ignored and the document editor is always used.

editor can also be an R function, in which case it is called with the arguments name, file, and title. Note that such a function will need to independently implement all desired functionality.

...

further arguments to be passed to or from methods.

Details

edit invokes the text editor specified by editor with the object name to be edited. It is a generic function, currently with a default method and one for data frames and matrices.

data.entry can be used to edit data, and is used by edit to edit matrices and data frames on systems for which data.entry is available.

It is important to realize that edit does not change the object called name. Instead, a copy of name is made and it is that copy which is changed. Should you want the changes to apply to the object name you must assign the result of edit to name. (Try fix if you want to make permanent changes to an object.)

In the form edit(name), edit deparses name into a temporary file and invokes the editor editor on this file. Quitting from the editor causes file to be parsed and that value returned. Should an error occur in parsing, possibly due to incorrect syntax, no value is returned. Calling edit(), with no arguments, will result in the temporary file being reopened for further editing.

Note that deparsing is not perfect, and the object recreated after editing can differ in subtle ways from that deparsed: see dput and .deparseOpts. (The deparse options used are the same as the defaults for dump.) Editing a function will preserve its environment. See edit.data.frame for further changes that can occur when editing a data frame or matrix.

Currently only the internal editor in Windows makes use of the title option; it displays the given name in the window header.

Extract data frame cell value

While working with tables you need to be able to select a specific data value from any row or column.

Recall that in Microsoft Excel, you can select a cell by specifying its location in the spreadsheet. For example, cell A1 represents column A and row 1. In data frames in R, the location of a cell is specified by row and column numbers.

Check out the different syntaxes which can be used for extracting data:

- **Extract value of a single cell**: `df_name[x, y]`, where `x` is the row number and `y` is the column number of a data frame called `df_name`.
- **Extract the entire row**: `df_name[x, ]`, where `x` is the row number. By not specifying the column number, we automatically choose all the columns for row `x`.
- **Extract the entire column**: `df_name[, y]` where `y` is the column number. By not specifying the row number, we automatically choose all the rows for column `y`.

Another way to extract data from `df_name` is by using the dollar sign in combination with the column names:

- `df_name$colname` refers to the entire column `col_name` in `df_name` data frame.
- `df_name$colname[x]` refers to row `x` of column `colname` in data frame `df_name`.

How To Merge Two DataFrames in R ?

We are going to see how to merge two R dataFrames. Merging of Data frames in R can be done in two ways.

- Merging columns
- Merging rows

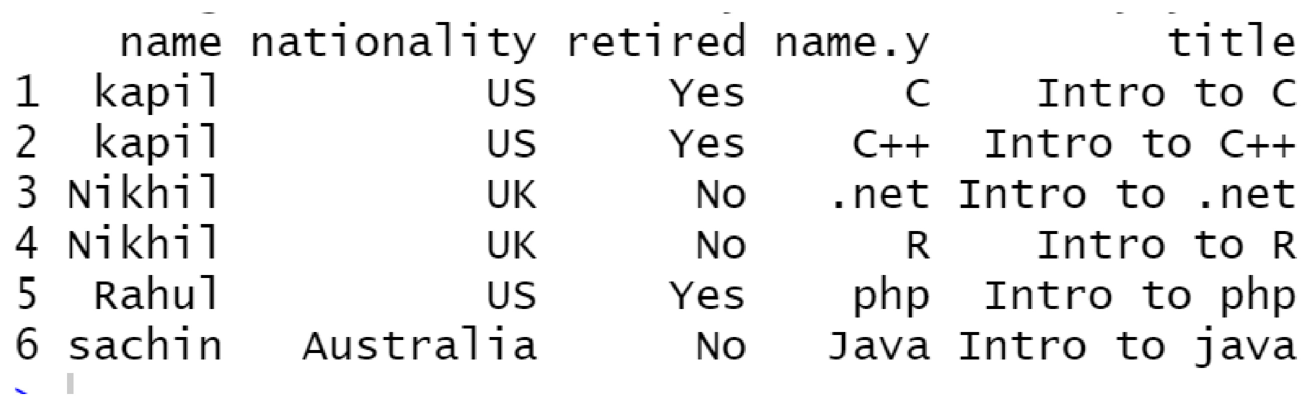
Merging columns

In this way, we merge the database horizontally. We use the merge function to merge two frames by one or more common key variables(i.e., an inner join).

```
dataframe_AB = merge(dataframe_A, dataframe_B, by="ID")
```

```
# merging two datasets

authors <- data.frame(

  name = c("kapil", "sachin", "Rahul","Nikhil","Rohan"),

  nationality = c("US","Australia","US","UK","India"),

  retired = c("Yes","No","Yes","No","No"))



books <-data.frame(

  name = c("C", "C++","Java","php",".net","R"),

  title = c("Intro to C","Intro to C++",

            "Intro to java", "Intro to php",

            "Intro to .net", "Intro to R"),

  author = c("kapil", "kapil","sachin", "Rahul",

             "Nikhil","Nikhil"))



  merge(authors, books, by.x = "name", by.y = "author")
```

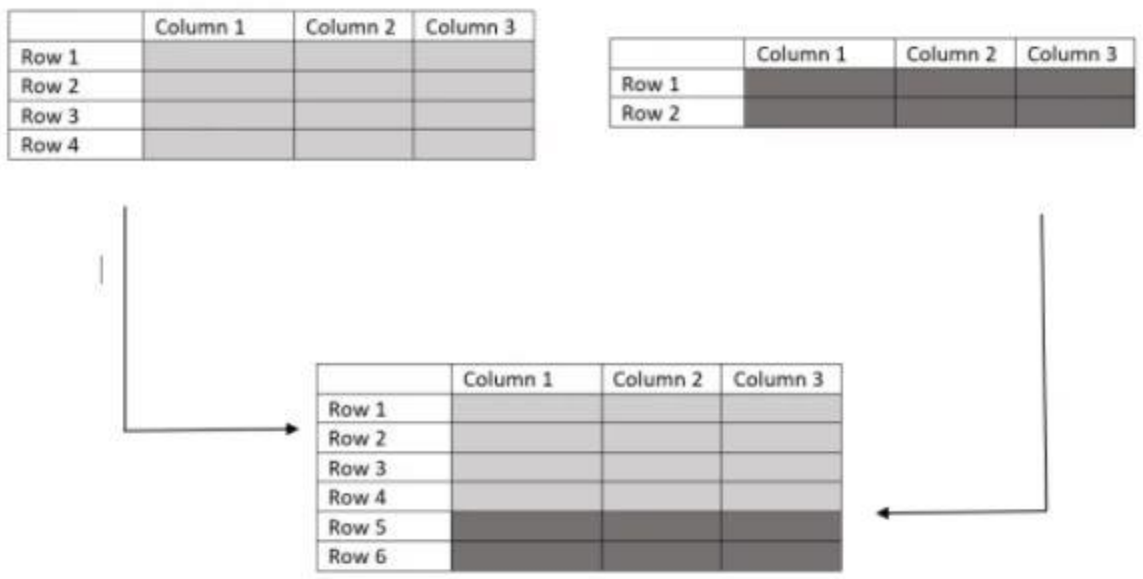
Output:



```
    name nationality retired name.y         title
1  kapil          US     Yes      C    Intro to C
2  kapil          US     Yes    C++  Intro to C++
3 Nikhil          UK      No   .net Intro to .net
4 Nikhil          UK      No      R    Intro to R
5  Rahul          US     Yes    php  Intro to php
6 sachin   Australia      No   Java Intro to java
```

Merging Rows
In this way, we merge the data frames vertically and use the rbind() function. rbind stands for row binding. The two data frames must have the same variables but need not be in the same order.

Note: If dataframe_A has variables that dataframe_B doesn’t have, either Delete the extra variables in dataframe_A or create the additional variables in dataframe_B and set them to NA.
As we can see from the below diagram, it combines rows of two dataframes.

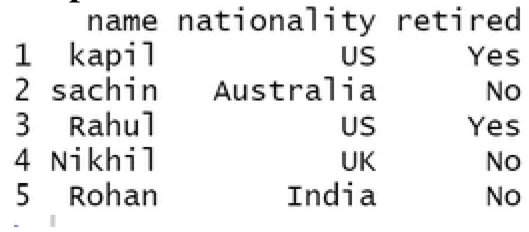


Below is the implementation:

- R

```
# merging two datasets

authors_A <- data.frame(

  name = c("kapil", "sachin", "Rahul"),

  nationality = c("US", "Australia", "US"),

  retired = c("Yes", "No", "Yes"))



authors_B <- data.frame(

  name = c("Nikhil", "Rohan"),

  nationality = c("UK", "India"),

  retired = c("No", "No"))



rbind(authors_A, authors_B)
```

Output:



```
    name nationality retired
1  kapil          US     Yes
2 sachin   Australia      No
3  Rahul          US     Yes
4 Nikhil          UK      No
5  Rohan       India      No
```

R melt() and cast() functions - Reshaping the data in R

we would be having a look at an important concept of R programming - **Reshaping data using R melt() and cast() functions**, in detail.

The R melt() and cast() functions help us to reshape the data within a data frame into any customized shape.

Working with the R melt() and cast() functions

Let's understand both the functions in detail. Here we go!

I. R melt() function

The `melt() function` in R programming is an in-built function. It enables us to reshape and elongate the data frames in a user-defined manner. It organizes the data values in a long data frame format.

Have a look at the below syntax!

Syntax:

```
melt(data-frame, na.rm = FALSE, value.name = “name”, id = 'columns')
```

We pass the data frame to the reshaped to the function along with na.rm = FALSE as the default value which means the NA values won't be ignored.

Further, we pass the new variable/column name to value.name parameter to store the elongated values obtained from the function into it.

The ID parameter is set to the column names of the data frame with respect to which the reshaping would happen.

Example:

In this example, we would be making use of libraries 'MASS, reshape2, and reshape'. Having created the data frame, we apply the melt() function on the data frame with respect to the column A and B.

```
rm(list = ls())
```

```
install.packages("MASS")
install.packages("reshape2")

install.packages("reshape")

library(MASS)

library(reshape2)

library(reshape)


A <- c(1,2,3,4,2,3,4,1)

B <- c(1,2,3,4,2,3,4,1)

a <- c(10,20,30,40,50,60,70,80)

b <- c(100,200,300,400,500,600,700,800)

data <- data.frame(A,B,a,b)

print("Original data frame:\n")

print(data)

melt_data <- melt(data, id = c("A","B"))

print("Reshaped data frame:\n")

print(melt_data)
```

Output:

```
[1]  "Original data frame:\n"

  A B  a   b

1  1  1  10  100

2  2  2  20  200

3  3  3  30  300

4  4  4  40  400

5  2  2  50  500

6  3  3  60  600
```

```
7  4  4  70  700

8  1  1  80  800


[1]  "Reshaped data frame:\n"

> print(melt_data)

   A B variable value

1   1 1        a    10

2   2 2        a    20

3   3 3        a    30

4   4 4        a    40

5   2 2        a    50

6   3 3        a    60

7   4 4        a    70

8   1 1        a    80

9   1 1        b   100

10  2 2        b   200

11  3 3        b   300

12  4 4        b   400

13  2 2        b   500

14  3 3        b   600

15  4 4        b   700

16  1 1        b   800
```

II. R cast() function

As seen above, after applying melt() function, the data frame gets converted to an elongated data frame. In order to regain the nearly original and natural shape of the data frame, `R cast() function` is used.

The cast() function accepts an aggregated function and a formula as a parameter *(here, formula is the manner in which the data is to be represented after reshaping)* and casts the elongated or molted data frame into a nearly aggregated form of data frame.

Syntax:

```
cast(data, formula, aggregate function)
```

We can provide the cast() function with any aggregate function available such as mean, sum, etc.

Example:

```
rm(list = ls())


library(MASS)

library(reshape2)

library(reshape)


A <- c(1,2,3,4,2,3,4,1)

B <- c(1,2,3,4,2,3,4,1)

a <- c(10,20,30,40,50,60,70,80)

b <- c(100,200,300,400,500,600,700,800)

data <- data.frame(A,B,a,b)


print("Original data frame:\n")

print(data)
```

```
melt_data <- melt(data, id = c("A"))


print("Reshaped data frame after melting:\n")

print(melt_data)


cast_data = cast(melt_data, A~variable, mean)


print("Reshaped data frame after casting:\n")

print(cast_data)
```

As seen above, we have passed mean as the aggregate function to cast() and have set variable equivalent to A variable as the format of representation.

Output:

```
[1]  "Original data frame:\n"

  A B  a   b

1  1  1  10  100

2  2  2  20  200

3  3  3  30  300

4  4  4  40  400

5  2  2  50  500

6  3  3  60  600

7  4  4  70  700

8  1  1  80  800


[1]  "Reshaped data frame after melting:\n"
```

```
   A variable value
1  1        B     1
2  2        B     2
3  3        B     3
4  4        B     4
5  2        B     2
6  3        B     3
7  4        B     4
8  1        B     1
9  1        a    10
10 2        a    20
11 3        a    30
12 4        a    40
13 2        a    50
14 3        a    60
15 4        a    70
16 1        a    80
17 1        b   100
18 2        b   200
19 3        b   300
20 4        b   400
21 2        b   500
22 3        b   600
23 4        b   700
24 1        b   800
```

```
[1]  "Reshaped data frame after casting:\n"

   A B  a   b

1  1  1  45  450

2  2  2  35  350

3  3  3  45  450

4  4  4  55  550
```

How to Use setwd and getwd in R?

We will discuss how to use setwd and getwd in the R programming language.

getwd() function

getwd() stands forget working directory. It is used to get the current working directory of the environment.

Syntax:

```
getwd()
```

We can also see the total number of files in the present working directory. For that, we have to use the length function.

Syntax:

```
length(list.files())
```

If we have to use the display the filenames, then the command is

```
list.files()
```

Example :

In this example, we will be using the getwd() function to get the current working directory.

```
getwd()
```

Output:

```
"C:/Users/Ramu/saisri"
```

Example :
Here, we will be using the getwd() function to get the length of the list of the files present in the working directory of the R console.

```
# get total file count

print(length(list.files()))



# get file names

print(list.files())
```

Output:
```
[1] 2

[1] "ai.R"   "ramu.R"
```

setwd() function

setwd() stands for set working directory. This is used to set the working environment.

Syntax:
```
setwd('path')
```

Example:
Here, we will be using the setwd() function to set the working directory.

```
setwd('C:/Ramu/saisri/')
```

dir R Function

Basic R Syntax:

```
dir(path)
```

The dir R function **returns a character vector of file and/or folder names within a directory**.

Example 1: Get File Names of Current Directory via getwd() & dir()

A typical application of the R dir Function is the extraction of file names within the currently used working directory. For this task, we have to use the getwd function first:

The getwd command returns the current working directory (i.e. the path of the folder in which we are currently working) as a character. By typing *path_cwd <- getwd()*, we store the current working directory in the data object *path_cwd*.

In your case, the path of the current working directory will of cause be different than mine. However, the code can be easily reproduced for your own directory.

Now, we can move on to the application of dir in R:

```
dir(path_cwd)                                  # Return file and folder names to console
# [1] "~$Upcoming Posts.xlsx"        "~$Statistical Programming Previous Posts.xlsx"
# [3] "R Code dir Function.R"        "Important Files"
```

We just have to insert the path of our directory into the dir() command. As you can see, the function returns a character vector that consists of all files and folders stored in this working directory.

In my case, the two xlsx files *Upcoming Posts.xlsx* and *Statistical Programming Previous Posts.xlsx* as well as the R code *R Code dir Function.R* and the folder *Important Files* are stored into the directory.

Note: You can distinguish files and folders by the ending of the name. My files end with *xlsx* and *R*, respectively, but the folder does not have any ending.

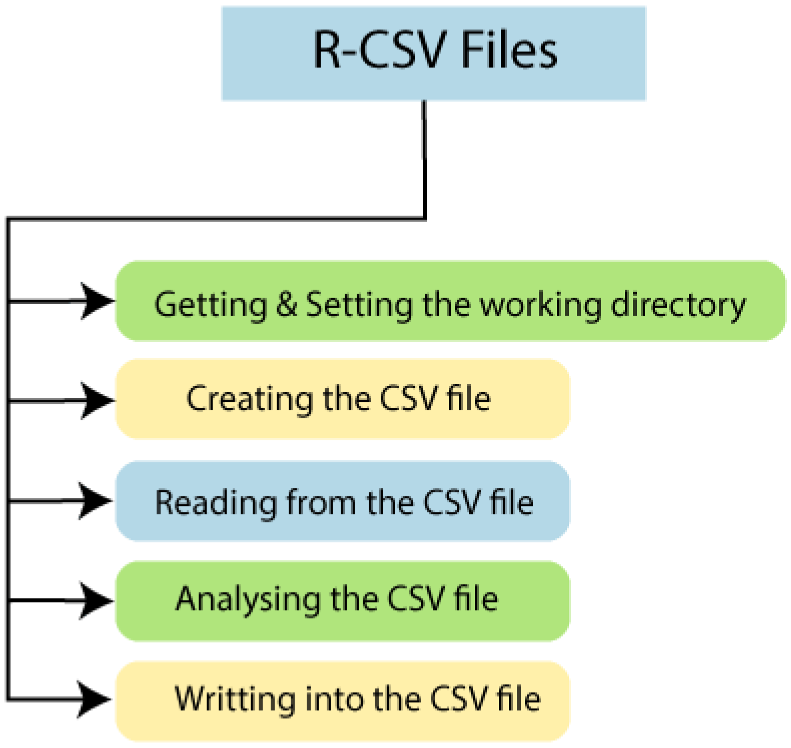
R CSV Files

A **Comma-Separated Values (CSV) file** is a plain text file which contains a list of data. These files are often used for the exchange of data between different applications. For example, databases and contact managers mostly support CSV files.

These files can sometimes be called **character-separated values** or **comma-delimited files**. They often use the comma character to separate data, but sometimes use other characters such as semicolons. The idea is that we can export the complex data from one application to a CSV file, and then importing the data in that CSV file to another application.

Storing data in excel spreadsheets is the most common way for data storing, which is used by the data scientists. There are lots of packages in R designed for accessing data from the excel spreadsheet. Users often find it easier to save their spreadsheets in comma-separated value files and then use R's built-in functionality to read and manipulate the data.

R allows us to read data from files which are stored outside the R environment. Let's start understanding how we can read and write data into CSV files. The file should be present in the current working directory so that R can read it. We can also set our directory and read file from there.


R-CSV Files
Getting & Setting the working directory
Creating the CSV file
Reading from the CSV file
Analysing the CSV file
Writting into the CSV file

Getting and setting the working directory

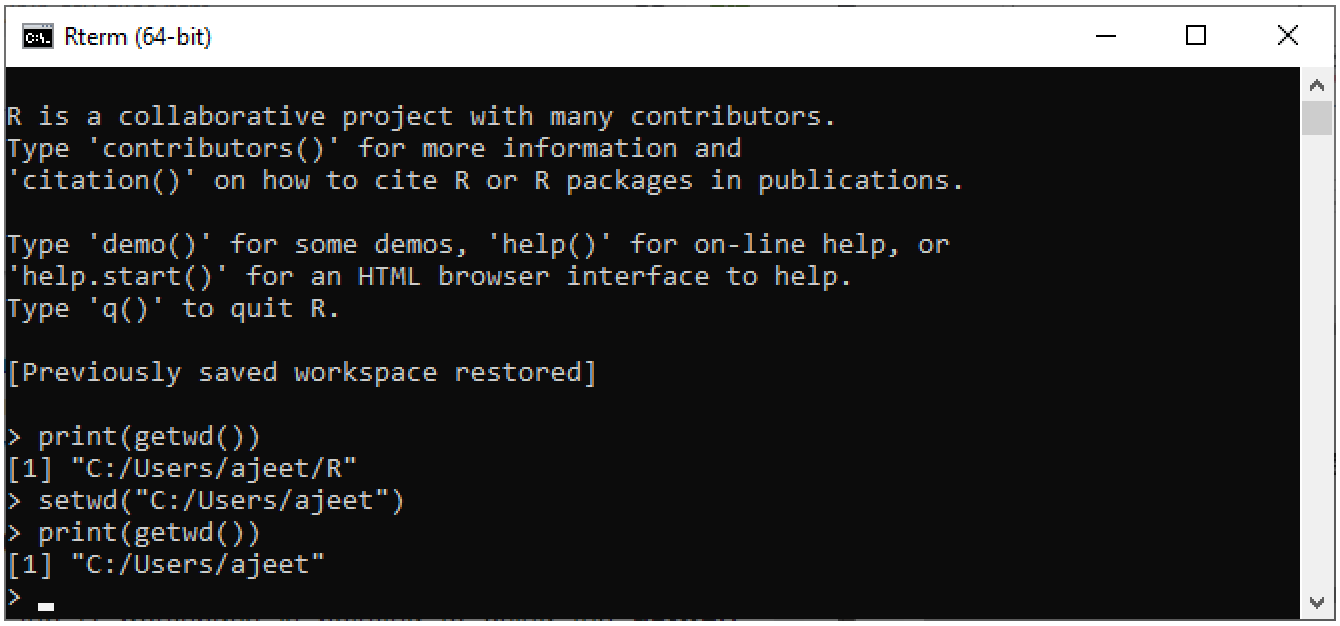
In R, getwd() and setwd() are the two useful functions. The getwd() function is used to check on which directory the R workspace is pointing. And the setwd() function is used to set a new working directory to read and write files from that directory.

Let's see an example to understand how getwd() and setwd() functions are used.

Example

1. # Getting and printing current working directory.
2. print(getwd())
3. # Setting the current working directory.
4. setwd("C:/Users/ajeet")
5. # Getting and printing the current working directory.
6. print(getwd())

Output



```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> print(getwd())
[1] "C:/Users/ajeet/R"
> setwd("C:/Users/ajeet")
> print(getwd())
[1] "C:/Users/ajeet"
> 
```

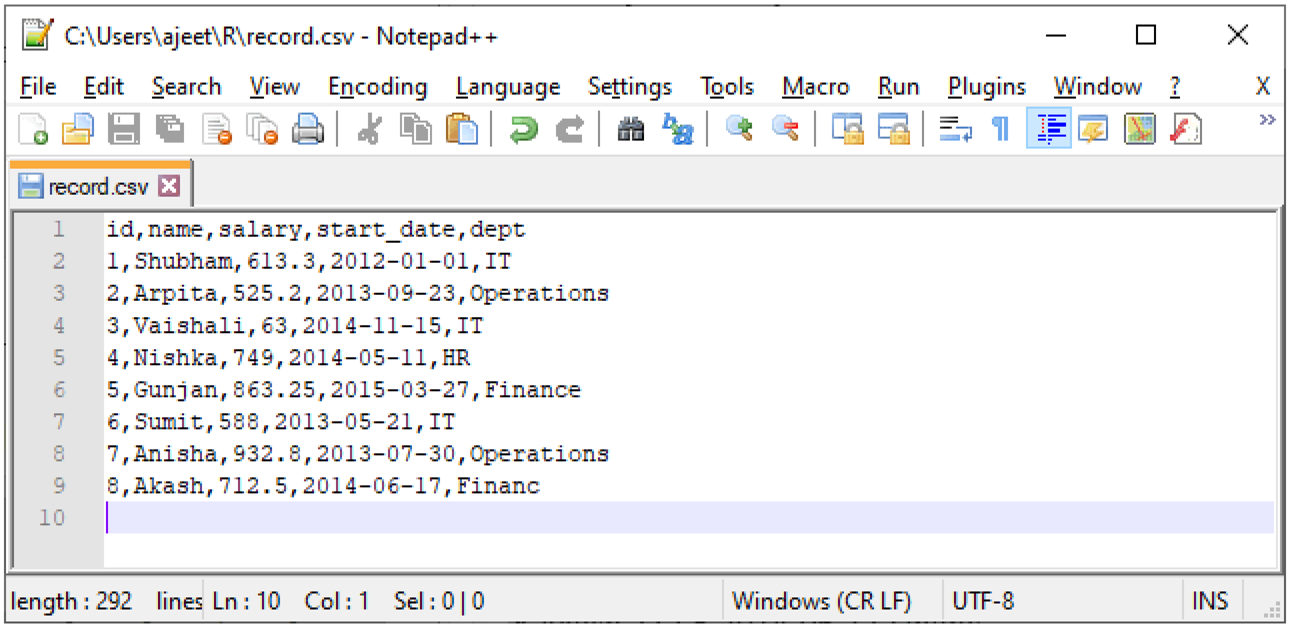
Creating a CSV File

A text file in which a comma separates the value in a column is known as a CSV file. Let's start by creating a CSV file with the help of the data, which is mentioned below by saving with .csv extension using the save As All files(*.*) option in the notepad.

Example: record.csv

```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
5,Gunjan,863.25,2015-03-27,Finance
6,Sumit,588,2013-05-21,IT
7,Anisha,932.8,2013-07-30,Operations
8,Akash,712.5,2014-06-17,Financ
```

Output


C:\Users\ajeet\R\record.csv - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
record.csv
length : 292 lines Ln : 10 Col : 1 Sel : 0 | 0
Windows (CR LF) UTF-8 INS

```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
5,Gunjan,863.25,2015-03-27,Finance
6,Sumit,588,2013-05-21,IT
7,Anisha,932.8,2013-07-30,Operations
8,Akash,712.5,2014-06-17,Financ
```

Reading a CSV file

R has a rich set of functions. R provides read.csv() function, which allows us to read a CSV file available in our current working directory. This function takes the file name as an input and returns all the records present on it.

Let's use our record.csv file to read records from it using read.csv() function.

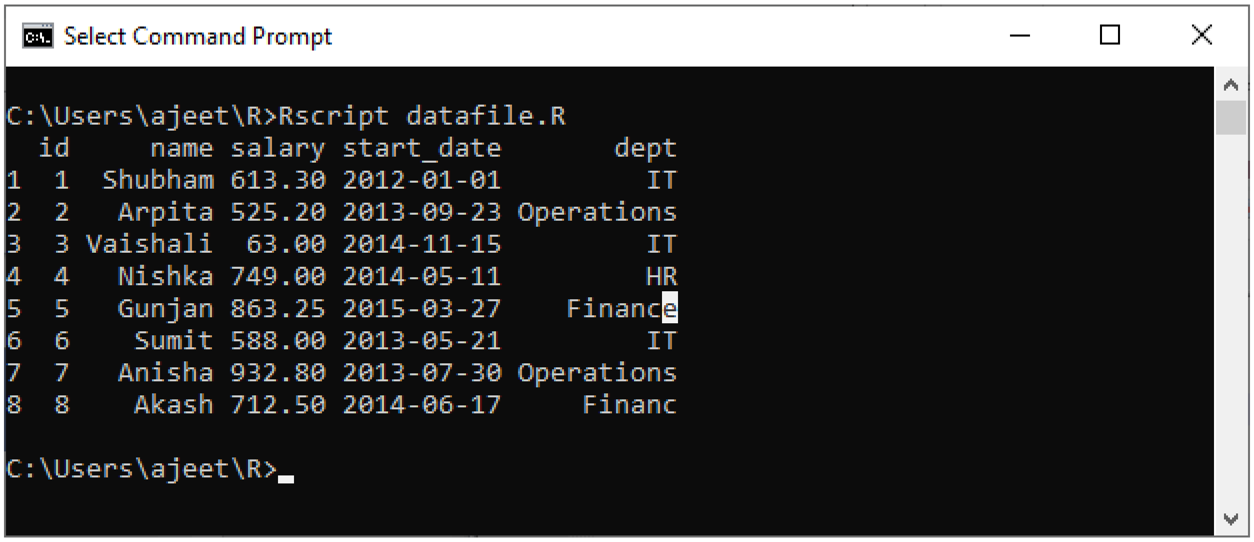
Example

```
data <- read.csv("record.csv")
```

2. print(data)

When we execute above code, it will give the following output

Output


Select Command Prompt

```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date       dept
1  1  Shubham 613.30 2012-01-01         IT
2  2   Arpita 525.20 2013-09-23 Operations
3  3 Vaishali  63.00 2014-11-15         IT
4  4   Nishka 749.00 2014-05-11         HR
5  5   Gunjan 863.25 2015-03-27    Finance
6  6    Sumit 588.00 2013-05-21         IT
7  7   Anisha 932.80 2013-07-30 Operations
8  8    Akash 712.50 2014-06-17     Financ

C:\Users\ajeet\R>_
```

Analyzing the CSV File

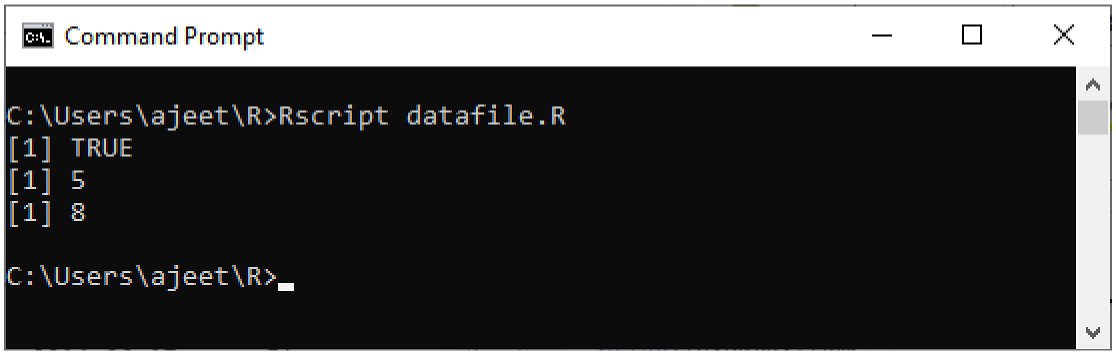
When we read data from the .csv file using **read.csv()** function, by default, it gives the output as a data frame. Before analyzing data, let's start checking the form of our output with the help of **is.data.frame()** function. After that, we will check the number of rows and number of columns with the help of **nrow()** and **ncol()** function.

Example

1. csv_data<- read.csv("record.csv")
2. print(is.data.frame(csv_data))
3. print(ncol(csv_data))
4. print(nrow(csv_data))

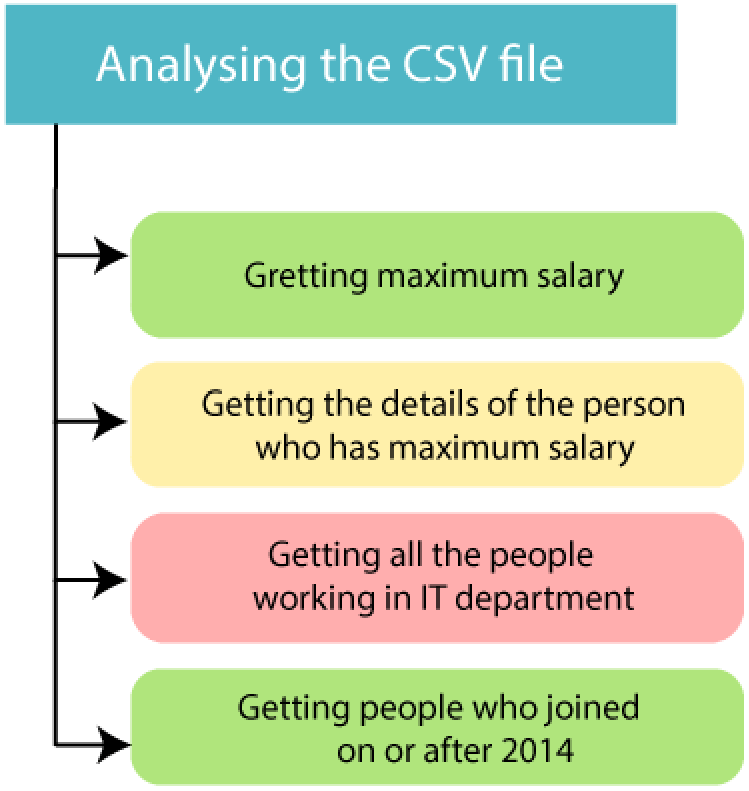
When we run above code, it will generate the following output:

Output



```
C:\Users\ajeet\R>Rscript datafile.R
[1] TRUE
[1] 5
[1] 8

C:\Users\ajeet\R>
```

From the above output, it is clear that our data is read in the form of the data frame. So we can apply all the functions of the data frame, which we have discussed in the earlier sections.

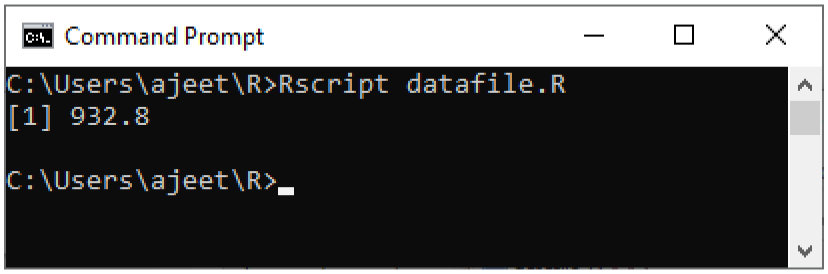

Analysing the CSV file
Gretting maximum salary
Getting the details of the person who has maximum salary
Getting all the people working in IT department
Getting people who joined on or after 2014

Example: Getting the maximum salary

1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
3.
4. # Getting the maximum salary from data frame.
5. max_sal<- max(csv_data$salary)

6. print(max_sal)

Output

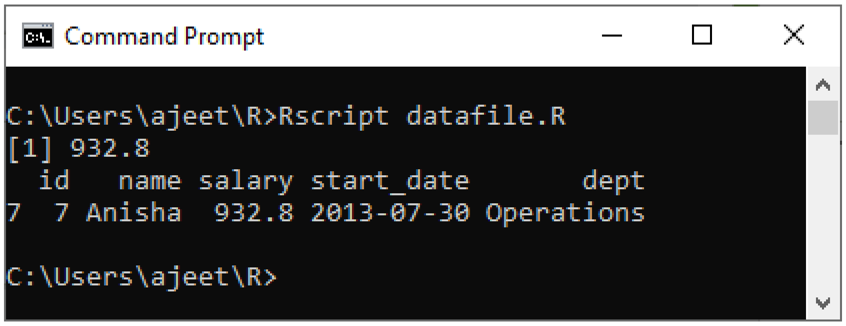


```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8

C:\Users\ajeet\R>_
```

Example: Getting the details of the person who have a maximum salary

```
# Creating a data frame.
csv_data<- read.csv("record.csv")

# Getting the maximum salary from data frame.
max_sal<- max(csv_data$salary)
print(max_sal)

#Getting the detais of the pweson who have maximum salary
details <- subset(csv_data,salary==max(salary))
print(details)
```

Output



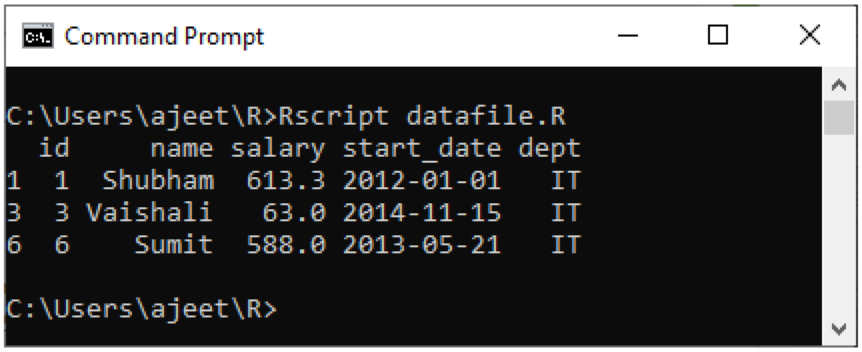
```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8
  id   name salary start_date       dept
7  7 Anisha  932.8 2013-07-30 Operations

C:\Users\ajeet\R>
```

Example: Getting the details of all the persons who are working in the IT department

```
# Creating a data frame.
csv_data<- read.csv("record.csv")
```

3.
4. #Getting the detais of all the pweson who are working in IT department
5. details <- subset(csv_data,dept=="IT")
6. print(details)

Output

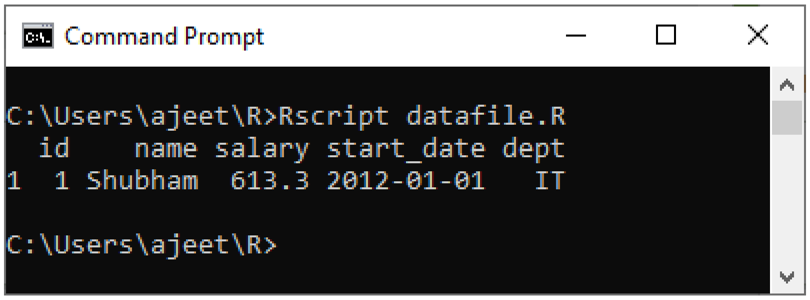


```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date dept
1  1  Shubham  613.3 2012-01-01   IT
3  3 Vaishali   63.0 2014-11-15   IT
6  6    Sumit  588.0 2013-05-21   IT

C:\Users\ajeet\R>
```

Example: Getting the details of the persons whose salary is greater than 600 and working in the IT department.

1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
3.
4. #Getting the detais of all the pweson who are working in IT department
5. details <- subset(csv_data,dept=="IT"&salary>600)
6. print(details)

Output



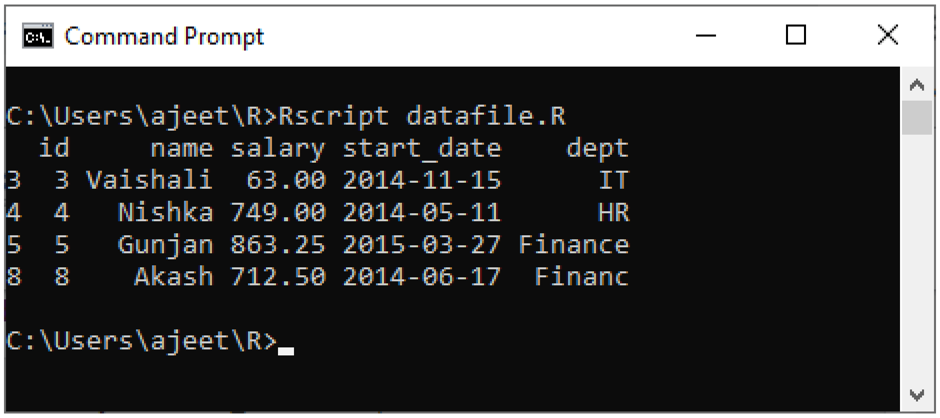
```
C:\Users\ajeet\R>Rscript datafile.R
  id    name salary start_date dept
1  1 Shubham  613.3 2012-01-01   IT

C:\Users\ajeet\R>
```

Example: Getting details of those peoples who joined on or after 2014.

1. # Creating a data frame.

```
csv_data<- read.csv("record.csv")

#Getting details of those peoples who joined on or after 2014
details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
print(details)
```

Output



```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date    dept
3  3 Vaishali  63.00 2014-11-15      IT
4  4   Nishka 749.00 2014-05-11      HR
5  5   Gunjan 863.25 2015-03-27 Finance
8  8    Akash 712.50 2014-06-17  Financ

C:\Users\ajeet\R>_
```

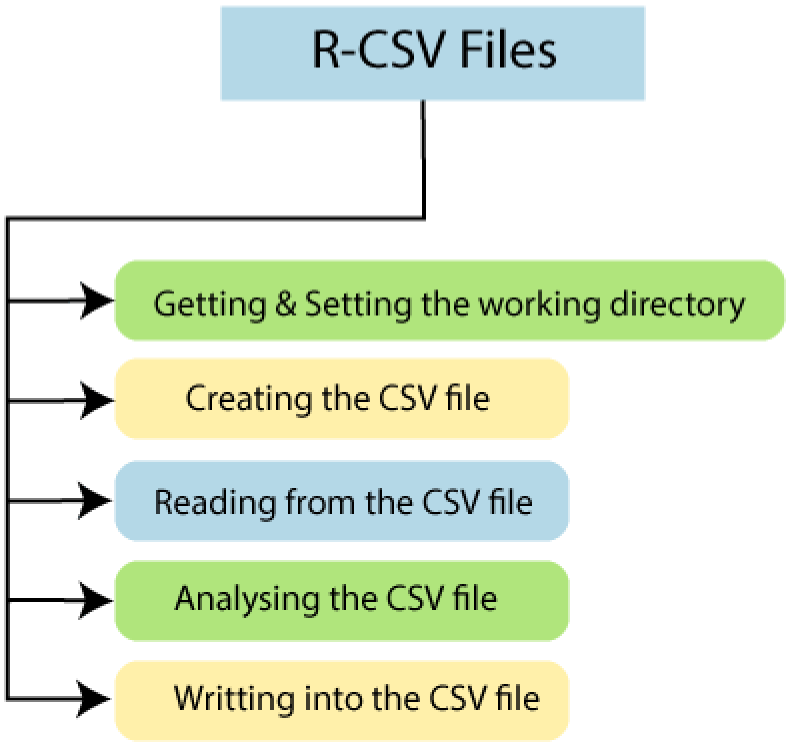
R CSV Files

A **Comma-Separated Values (CSV) file** is a plain text file which contains a list of data. These files are often used for the exchange of data between different applications. For example, databases and contact managers mostly support CSV files.

These files can sometimes be called **character-separated values** or **comma-delimited files**. They often use the comma character to separate data, but sometimes use other characters such as semicolons. The idea is that we can export the complex data from one application to a CSV file, and then importing the data in that CSV file to another application.

Storing data in excel spreadsheets is the most common way for data storing, which is used by the data scientists. There are lots of packages in R designed for accessing data from the excel spreadsheet. Users often find it easier to save their spreadsheets in comma-separated value files and then use R's built-in functionality to read and manipulate the data.

R allows us to read data from files which are stored outside the R environment. Let's start understanding how we can read and write data into CSV files. The file should be present in the current working directory so that R can read it. We can also set our directory and read file from there.


R-CSV Files
Getting & Setting the working directory
Creating the CSV file
Reading from the CSV file
Analysing the CSV file
Writting into the CSV file

Getting and setting the working directory

In R, getwd() and setwd() are the two useful functions. The getwd() function is used to check on which directory the R workspace is pointing. And the setwd() function is used to set a new working directory to read and write files from that directory.

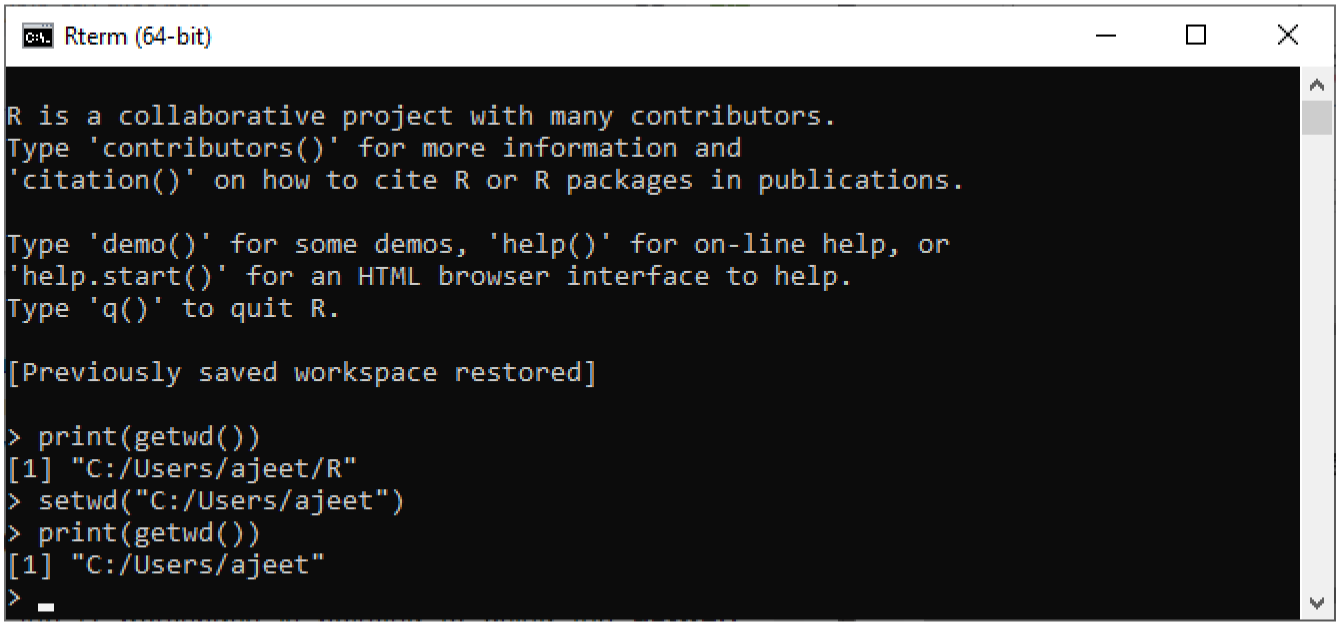
Let's see an example to understand how getwd() and setwd() functions are used.

Example

```
# Getting and printing current working directory.
print(getwd())
```

```
# Setting the current working directory.
setwd("C:/Users/ajeet")
# Getting and printingthe current working directory.
print(getwd())
```

Output



```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> print(getwd())
[1] "C:/Users/ajeet/R"
> setwd("C:/Users/ajeet")
> print(getwd())
[1] "C:/Users/ajeet"
> 
```

Creating a CSV File

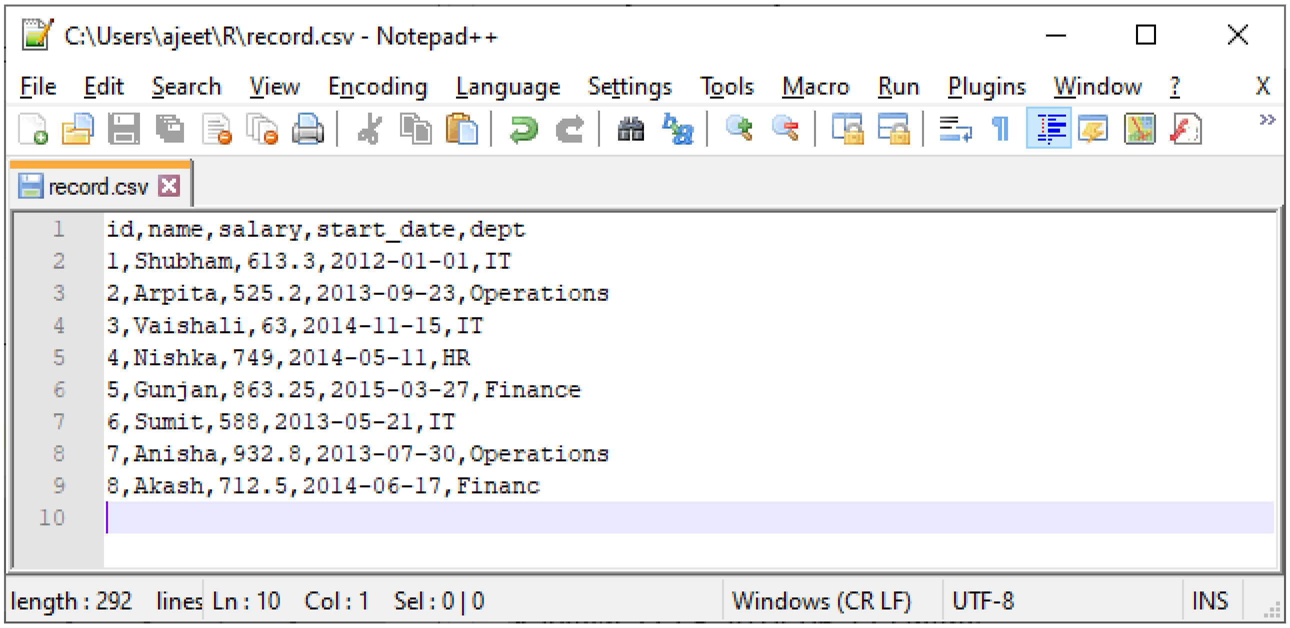
A text file in which a comma separates the value in a column is known as a CSV file. Let's start by creating a CSV file with the help of the data, which is mentioned below by saving with .csv extension using the save As All files(*.*) option in the notepad.

Example: record.csv

```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
5,Gunjan,863.25,2015-03-27,Finance
6,Sumit,588,2013-05-21,IT
7,Anisha,932.8,2013-07-30,Operations
```

8,Akash,712.5,2014-06-17,Financ

Output


C:\Users\ajeet\R\record.csv - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
record.csv

```
id,name,salary,start_date,dept
1,Shubham,613.3,2012-01-01,IT
2,Arpita,525.2,2013-09-23,Operations
3,Vaishali,63,2014-11-15,IT
4,Nishka,749,2014-05-11,HR
5,Gunjan,863.25,2015-03-27,Finance
6,Sumit,588,2013-05-21,IT
7,Anisha,932.8,2013-07-30,Operations
8,Akash,712.5,2014-06-17,Financ
```

length : 292 lines Ln : 10 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Reading a CSV file

R has a rich set of functions. R provides read.csv() function, which allows us to read a CSV file available in our current working directory. This function takes the file name as an input and returns all the records present on it.

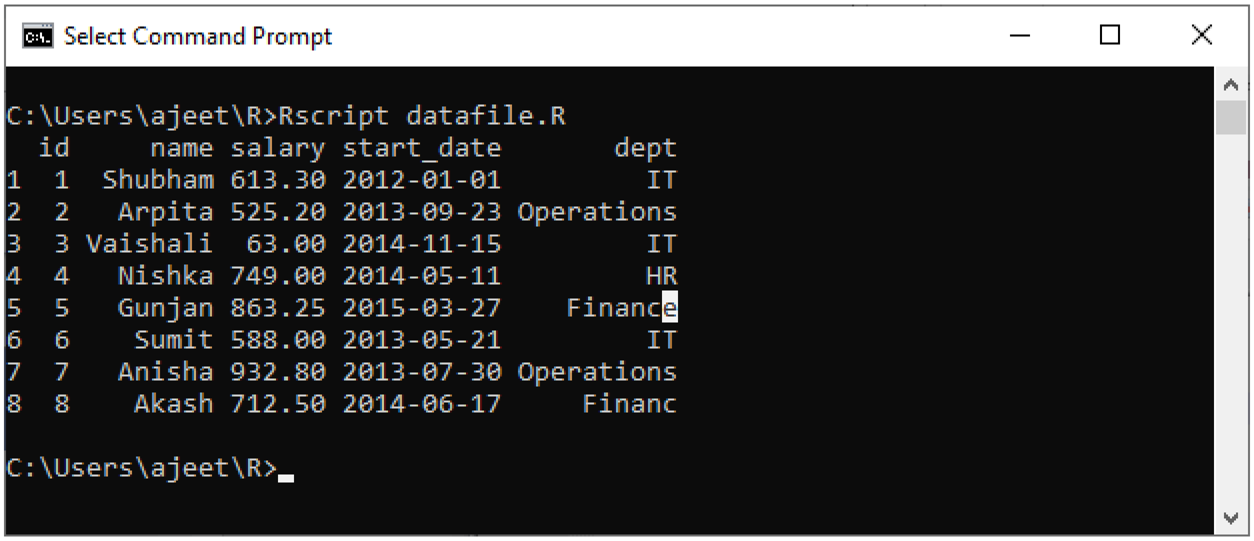
Let's use our record.csv file to read records from it using read.csv() function.

Example

```
data <- read.csv("record.csv")
print(data)
```

When we execute above code, it will give the following output

Output



```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date       dept
1  1  Shubham 613.30 2012-01-01         IT
2  2   Arpita 525.20 2013-09-23 Operations
3  3 Vaishali  63.00 2014-11-15         IT
4  4   Nishka 749.00 2014-05-11         HR
5  5   Gunjan 863.25 2015-03-27    Finance
6  6    Sumit 588.00 2013-05-21         IT
7  7   Anisha 932.80 2013-07-30 Operations
8  8    Akash 712.50 2014-06-17     Financ

C:\Users\ajeet\R>
```

Analyzing the CSV File

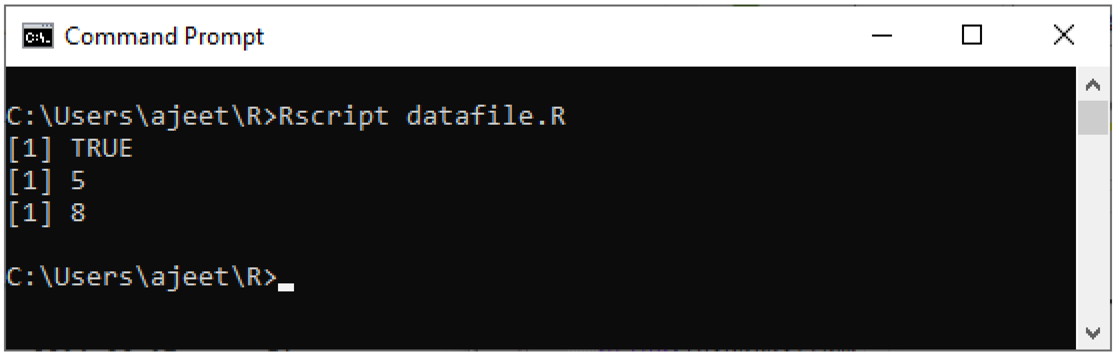
When we read data from the .csv file using **read.csv()** function, by default, it gives the output as a data frame. Before analyzing data, let's start checking the form of our output with the help of **is.data.frame()** function. After that, we will check the number of rows and number of columns with the help of **nrow()** and **ncol()** function.

Example

```
csv_data<- read.csv("record.csv")
print(is.data.frame(csv_data))
print(ncol(csv_data))
print(nrow(csv_data))
```

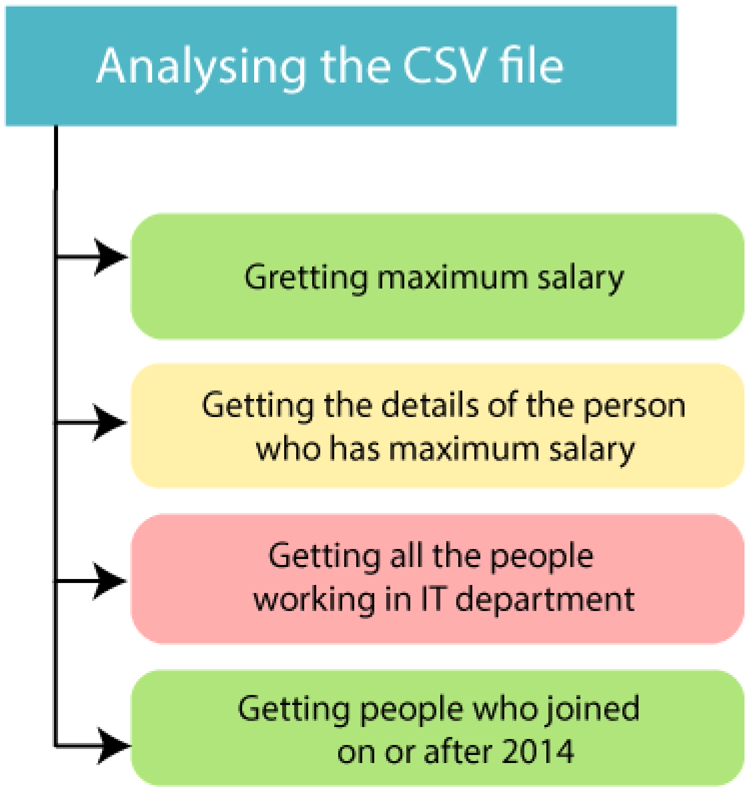
When we run above code, it will generate the following output:

Output



```
C:\Users\ajeet\R>Rscript datafile.R
[1] TRUE
[1] 5
[1] 8

C:\Users\ajeet\R>
```

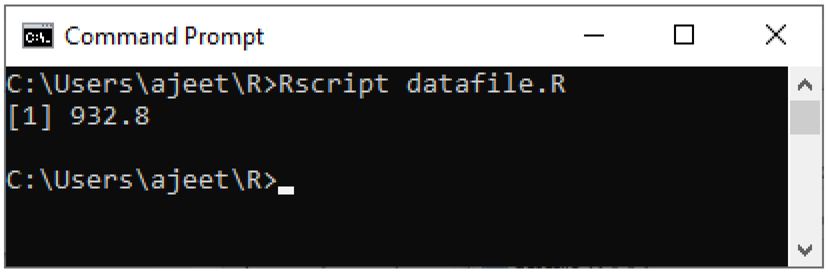
From the above output, it is clear that our data is read in the form of the data frame. So we can apply all the functions of the data frame, which we have discussed in the earlier sections.


Analysing the CSV file
Gretting maximum salary
Getting the details of the person who has maximum salary
Getting all the people working in IT department
Getting people who joined on or after 2014

Example: Getting the maximum salary

```
# Creating a data frame.
csv_data<- read.csv("record.csv")

# Getting the maximum salary from data frame.
max_sal<- max(csv_data$salary)
print(max_sal)
```

Output



```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8

C:\Users\ajeet\R>
```

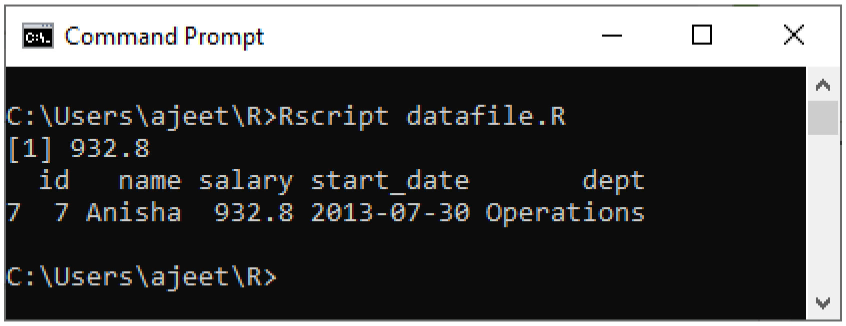
Example: Getting the details of the person who have a maximum salary

# Creating a data frame.
csv_data<- read.csv("record.csv")

# Getting the maximum salary from data frame.
max_sal<- max(csv_data$salary)
print(max_sal)

#Getting the detais of the pweson who have maximum salary
details <- subset(csv_data,salary==max(salary))
print(details)

Output



```
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8
  id   name salary start_date       dept
7  7 Anisha  932.8 2013-07-30 Operations

C:\Users\ajeet\R>
```

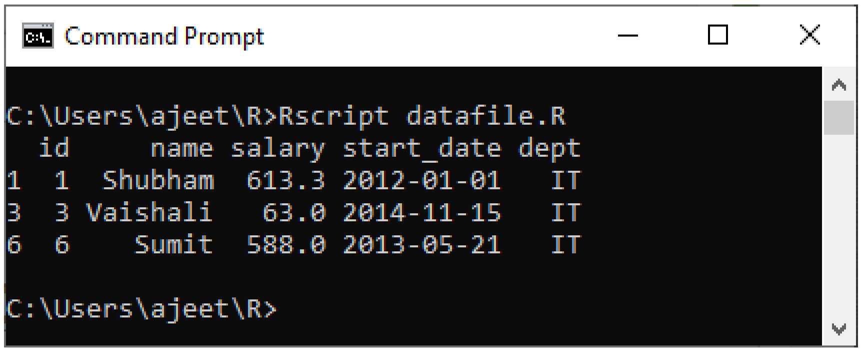
Example: Getting the details of all the persons who are working in the IT department

# Creating a data frame.
csv_data<- read.csv("record.csv")

#Getting the detais of all the pweson who are working in IT department
details <- subset(csv_data,dept=="IT")

print(details)

Output



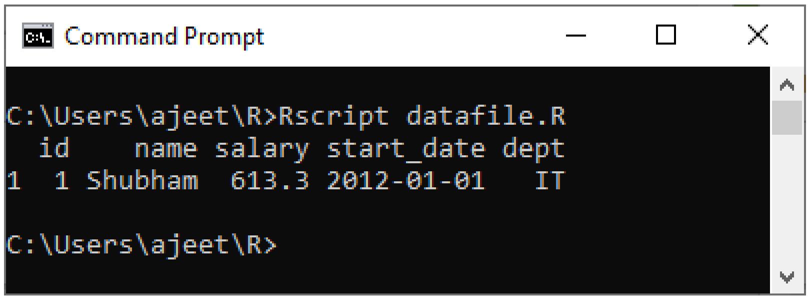
```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date dept
1  1  Shubham  613.3 2012-01-01   IT
3  3 Vaishali   63.0 2014-11-15   IT
6  6    Sumit  588.0 2013-05-21   IT

C:\Users\ajeet\R>
```

Example: Getting the details of the persons whose salary is greater than 600 and working in the IT department.

# Creating a data frame.
csv_data<- read.csv("record.csv")

#Getting the detais of all the pweson who are working in IT department
details <- subset(csv_data,dept=="IT"&salary>600)
print(details)

Output

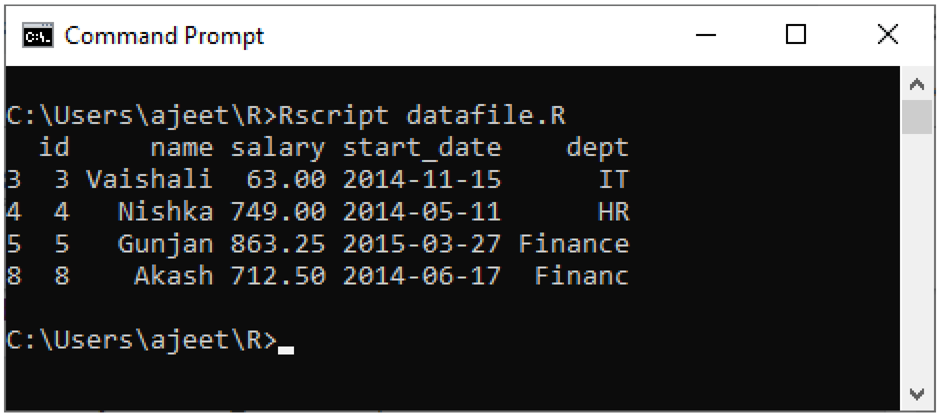


```
C:\Users\ajeet\R>Rscript datafile.R
  id    name salary start_date dept
1  1 Shubham  613.3 2012-01-01   IT

C:\Users\ajeet\R>
```

Example: Getting details of those peoples who joined on or after 2014.
# Creating a data frame.
csv_data<- read.csv("record.csv")

#Getting details of those peoples who joined on or after 2014
details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
print(details)

Output


Command Prompt

```
C:\Users\ajeet\R>Rscript datafile.R
  id     name salary start_date    dept
3  3 Vaishali  63.00 2014-11-15      IT
4  4   Nishka 749.00 2014-05-11      HR
5  5   Gunjan 863.25 2015-03-27 Finance
8  8    Akash 712.50 2014-06-17  Financ

C:\Users\ajeet\R>
```

Writing into a CSV file

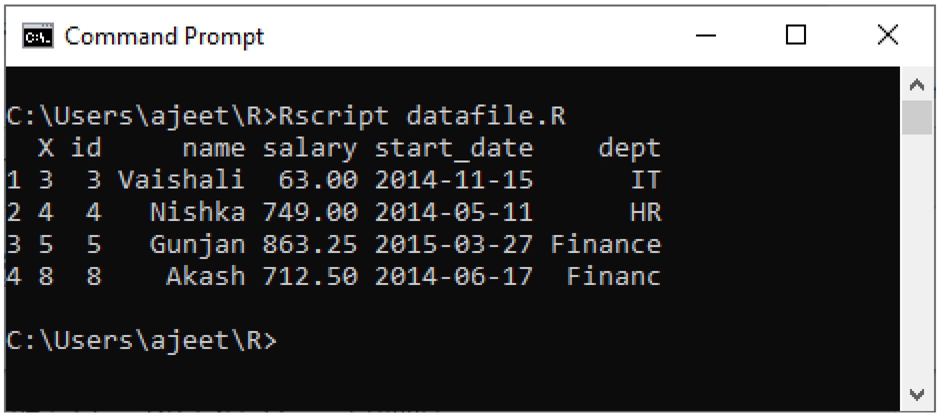
Like reading and analyzing, R also allows us to write into the .csv file. For this purpose, R provides a write.csv() function. This function creates a CSV file from an existing data frame. This function creates the file in the current working directory.

Let's see an example to understand how **write.csv()** function is used to create an output CSV file.

Example

```
csv_data<- read.csv("record.csv")

#Getting details of those peoples who joined on or after 2014
details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))

# Writing filtered data into a new file.
write.csv(details,"output.csv")
new_details<- read.csv("output.csv")
print(new_details)
```

Output



```
C:\Users\ajeet\R>Rscript datafile.R
  X id     name salary start_date    dept
1 3  3 Vaishali  63.00 2014-11-15      IT
2 4  4   Nishka 749.00 2014-05-11      HR
3 5  5   Gunjan 863.25 2015-03-27 Finance
4 8  8    Akash 712.50 2014-06-17  Financ

C:\Users\ajeet\R>
```

More on CSV File

How to Read CSV File into DataFrame in R

How do I read data from a CSV file into R DataFrame? Use read.csv() function in R to import a CSV file into a DataFrame. CSV file format is the easiest way to store scientific, analytical, or any structured data (two-dimensional with rows and columns). Data in CSV is separated by delimiter most commonly comma (,) but you can also use any character like pipe, tab e.t.c

In this article, I will explain how to read a CSV file into DataFrame and also explain different options you can use while reading a CSV effectively without errors. In order to write or export a CSV file use write.csv().

1. Quick Examples of Read CSV File in R

The following are quick examples of how to import a CSV in R by using the read.csv() function and its optional arguments.

```
# Quick examples


# Read CSV into DataFrame

read_csv = read.csv('/Users/admin/file.csv')


# Read with custom delimiter

read_csv = read.csv('/Users/admin/file.csv',sep=',')
```

```
# Read without header
read_csv = read.csv('/Users/admin/file_noheader.csv', header=FALSE)


# Set Column Names
colnames(read_csv) = c('id','name','dob','gender')
str(read_csv)


# Replaces all -1 and empty string as <NA>
read_csv = read.csv('/Users/admin/file.csv',na.strings=c(-1,''))


# Keep String as Character.
read_csv = read.csv('/Users/admin/file_noheader.csv', stringsAsFactors='FALSE')


# Use UTF-8 encoding
read_csv = read.csv('/Users/admin/file_noheader.csv', encoding='utf-8')
```

2. Read CSV File in R

In order to read a CSV file in R use its base function `read.csv()`, which loads the data from the CSV file into DataFrame. Once the data frame was created and to perform operations refer to R data frame tutorial for examples.

Following is the syntax of the `read.csv()` function in R. Note that CSV and Excel files are different hence to load an excel file in R use packages like `readxl`, `xlsx`, and `openxlsx`

```
# Syntax of read.csv()
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

Copy

Let’s read a comma separate CSV file into a DataFrame.

```
# Read CSV into DataFrame
read_csv = read.csv('/Users/admin/file.csv')
print(read_csv)
```

Yields below output.

```
# Output
  id name        dob gender
1 10  sai 1990-10-02      M
2 NA  ram 1981-03-24
3 -1 <NA> 1987-06-14      F
4 13      1985-08-16   <NA>
```

3. Read CSV with Custom Delimiter using sep Argument

By default read.csv() function uses a comma delimiter however, you can use any custom delimiter by using `sep` argument. For example, use `sep='|'` to read a CSV file with data separated by a pipe, for tab use `sep='\t'`.

```
# Usage of sep param
read_csv = read.csv('/Users/admin/file.csv',sep=',')
print(read_csv)
```

Copy

4. Read CSV without Header using header Argument

Sometimes you may receive the CSV file without a header row (column names), if you receive such a file, use the `header` argument with `FALSE` to not consider the first record in a CSV file as a header. By default `header` param is set to a value `TRUE` hence, it automatically considers the first record in a CSV file as a header.

Let’s take another CSV file `file_noheader.csv` without a header row (column names) and load into DataFrame.

```
# Use header=False

read_csv = read.csv('/Users/admin/file_noheader.csv', header=FALSE)

print(read_csv)
```

Yields below output.

```
# Output

  V1   V2         V3     V4

1 10  sai 1990-10-02      M

2 NA  ram 1981-03-24

3 -1 <NA> 1987-06-14      F

4 13      1985-08-16   <NA>
```

Note that the default column names it assigns as V1, V2, V3, and V4. To rename columns on DataFrame to your own use `colnames()`.

```
# Set column names

colnames(read_csv) = c('id','name','dob','gender')

print(read_csv)
```

5. Usage of na.strings Argument

When you are working with large or small files, you often get missing or unexpected data in certain cells of rows & columns. Usually, these missing data are represented as empty. If you notice our DataFrame result from the above outputs, you would see some missing values like an empty string on `name`, `gender`, and -1 unexpect value for `id` column.

By using na.strings argument, you can specify vector of values you would like to consider as NA. In the below example, I have used c(-1,'') to instruct read.csv() method to consider all -1 and empty strings as NA. You can also replace an empty string with NA on the DataFrame.

```
# Replaces all -1 and empty string as <NA>
read_csv = read.csv('/Users/admin/file.csv',na.strings=c(-1,''))
print(read_csv)
```

Yields below output.

```
# Output
  id name        dob gender
1 10  sai 1990-10-02      M
2 NA  ram 1981-03-24   <NA>
3 NA <NA> 1987-06-14      F
4 13 <NA> 1985-08-16   <NA>
```

Sometimes you would also be required to replace NA values with 0 on numeric columns on DataFrame.

6. Usage of stringsAsFactors Argument

If you are using an older version which is prior to R 4.0, all columns that have character string data are by default converted to factor types. When a column is in factor type, you can't perform many string operations hence, to keep string columns as character type use stringsAsFactors=FALSE while reading a CSV file in R.

With a newer version on R, you don't have to use this argument as R by default considers character data as a string. I am using R version 4.0, hence all my string columns are converted to character (chr) type. Use str() to display the structure of the DataFrame.

```
# Keep String as Character.
read_csv = read.csv('/Users/admin/file_noheader.csv', stringsAsFactors='FALSE')
```

```
str(read_csv)
```

Yields below output.

```
# Output
# I am using R new version hence string is in chr type
'data.frame': 4 obs. of  4 variables:
 $ id    : int  10 11 12 13
 $ name  : chr  "sai" "ram" NA "sahithi"
 $ dob   : chr  "1990-10-02" "1981-03-24" "1987-06-14" "1985-08-16"
 $ gender: chr  "M" "" "F" "F"
```

Copy

7. CSV encoding

If you receive a CSV file with other encodings, for example having Spanish characters e.t.c, you should use `encoding` param with the appropriate encoding. To read it as UTF-8, use `encoding=UTF-8` argument while importing a file into DataFrame.

```
# Use UTF-8 encoding
read_csv = read.csv('/Users/admin/file_noheader.csv', encoding='utf-8')
print(read_csv)
```

Copy

8. read.csv2()

`read.csv2()` is another R function to import CSV file into DataFrame. This function by default uses a comma as a decimal point and a semicolon as a field separator.

```
# Using read_csv()
```

```
read_csv = read.csv2('/Users/admin/file_noheader.csv')

print(read_csv)
```

Copy

9. Import CSV using read.table()

To import a CSV file in R use `read.table()`, which doesn’t use any default delimiter. You need to explicitly specify what delimiter and how you wanted to read a CSV file. Functions read.csv(), read.csv2() are wrappers and uses read.table() internally.

```
# Using read.table()

read_csv = read.table('/Users/admin/file.csv',sep=',')

print(read_csv)
```

Copy

10. Use read_csv()

If you are working with larger files, you should use the `read_csv()` function `readr` package. `readr` is a third-party library hence, in order to use `readr` library, you need to first install it by using `install.packages('readr')`. Once installation completes, load the `readr` library in order to use this `read_csv()` method. To load a library in R use `library("readr")`.

```
# Load readr

library("readr")


# Read CSV into DataFrame

read_csv = read_csv('/Users/admin/file.csv')

print(read_csv)
```

How to Use min() and max() in R

Finding min and max values is pretty much simple with the functions **min()** and **max()** in R.

You know that we have functions like mean, median, sd, and mode to calculate the average, middle, and dispersion of values respectively. But did you ever thought of a function which gives you min and max values in a vector or a data frame?

If so, congratulations, you have got functions named **min()** and **max()** which returns the minimum and maximum values respectively.

Sounds interesting right? Let's see how it works!

Let's start with the syntax

The syntax of the min() function is given below.

```
min(x, na.rm = FALSE)
```

- **x =** vector or a data frame.
- **na.rm =** remove NA values, if it mentioned False it considers NA or if it mentioned True it removes NA from the vector or a data frame.

The syntax of the max() function is given below.

```
max(x, na.rm = FALSE)
```

Copy

- **x =** vector or a data frame.
- **na.rm =** remove NA values, if it mentioned False it considers NA or if it mentioned True it removes NA from the vector or a data frame.

Max() function in R

In this section, we are going to find the max values present in the vector. For this, we first create a vector and then apply the max() function, which returns the max value in the vector.

```
#creates a vector
vector<-c(45.6,78.8,65.0,78.9,456.7,345.89,87.6,988.3)
```

```
#returns the max values present in the vector
max(vector)
```

Copy

Output= 988.3

Min() function in R

Here, we are going to find the minimum value in a vector using function min(). You can create a vector and then apply min() to the vector which returns the minimum value as shown below.

```
#creates a vector
vector<-c(45.6,78.8,65.0,78.9,456.7,345.89,87.6,988.3)

#returns the minimum value present in the vector
min(vector)
```

Copy

Output = 45.6

Max() function in R with NA values

Sometimes in the data analysis, you may encounter the NA values in a data frame as well as a vector. Then you need to bypass the NA values in order to get the desired result.

The max function won't return any values if it encounters the NA values in the process. Hence you have to remove NA values from the vector or a data frame to get the **max** value.

```
#creates a vector having NA values
df<- c(134,555,NA,567,876,543,NA,456)

#max function won't return any value because of the presence of NA.
max(df)

#results in NA instead of max value
```

```
Output = NA
```
Copy

So to avoid this and get the max value we are using **na.rm** function to remove NA values from the vector. Now you can see that the max() function is returning the maximum value.

```
#max function with remove NA values is set as TRUE.
max(df, na.rm = T)
```
Copy

Output = 876

Min() function in R with NA values

Just like we applied the max function in the above section, here we are going to find the minimum value in a vector having NA values.

```
#creates a vector having NA values
df<- c(134,555,NA,567,876,543,NA,456)

#returns NA instead of minimum value
min(df)
```
Copy

Output = NA

To overcome this, we are using **na.rm** function to remove NA values from the vector. Now you can that the min() function is returning the min value.

```
#creates a vector having NA values
df<- c(134,555,NA,567,876,543,NA,456)

#removes NA values and returns the minimum value in the vector
min(df, na.rm = T)
```
Copy

Output = 134

Min() and Max() functions in a character vector

Till now we have dealt with numerical minimum and maximum values. If I have to tell you something, I wish to say that you can also find the min and max values for a character vector as well. Yes, you heard it right!

Let’s see how it works!

In the case of character vectors, the min and max functions will consider alphabetical order and returns min and max values accordingly as shown below.

```
#creates a character vector with some names
character_vector<- c('John','Angelina','Smuts','Garena','Lucifer')

#returns the max value
max(character_vector)
```
Copy

Output = “Smuts”

Similarly, we can find the minimum values in the character vector as well using min() function which is shown below.

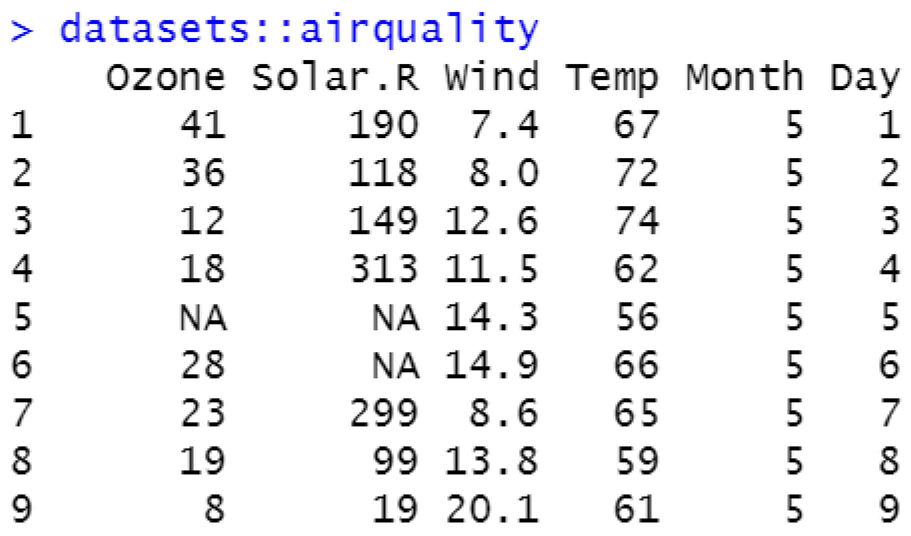
```
#creates a character vector with some names
character_vector<- c('John','Angelina','Smuts','Garena','Lucifer')

#returns the minimum values in the vector
min(character_vector)
```
Copy

Output = “Angelina”

Min() and Max() functions in a data frame

Let’s find the minimum and maximum values of a data frame by importing it. The min and max values in a dataset will give a fair idea about the data distribution.

This is the air quality dataset that is available in R studio. Note that the dataset includes NA values. With the knowledge of removing NA values using **na.rm** function, let’s find the min and max values in the **Ozone values**.



```
> datasets::airquality
  Ozone Solar.R Wind Temp Month Day
1     41     190  7.4   67     5   1
2     36     118  8.0   72     5   2
3     12     149 12.6   74     5   3
4     18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
7     23     299  8.6   65     5   7
8     19      99 13.8   59     5   8
9      8      19 20.1   61     5   9
```

```
min(airquality$Ozone, na.rm=T)
```

Copy

Output = 1

```
max(airquality$Ozone, na.rm = T)
```

Copy

Output = 168

Let's find the min and max values of the Temperature values in the airquality dataset.

```
 min(airquality$Temp, na.rm = T)
```

Copy

Output = 56

```
max(airquality$Temp, na.rm = T)
```

Copy

Output = 97

How to Find Standard Deviation in R?

So what is the standard deviation?

- **'Standard deviation is the measure of the dispersion of the values'.**
- The higher the standard deviation, the wider the spread of values.
- The lower the standard deviation, the narrower the spread of values.

- In simple words the formula is defined as - **Standard deviation is the square root of the 'variance'.**

Importance on Standard deviation

Standard deviation is very popular in the statistics, but why? the reasons for its popularity and its importance are listed below.

- Standard deviation converts the negative number to a positive number by **squaring** it.
- It shows the **larger deviations** so that you can particularly look over them.
- It shows the **central tendency,** which is a very useful function in the analysis.
- It has a major role to play in **finance, business, analysis, and measurements.**

Before we roll into the topic, keep this definition in your mind!

Variance - It is defined as the squared differences between the observed value and expected value.

Find the Standard deviation in R for values in a list

In this method, we will create a list 'x' and add some value to it. Then we can find the standard deviation of those values in the list.

```
 x <- c(34,56,87,65,34,56,89)    #creates list 'x' with some values in it.

 sd(x)  #calculates the standard deviation of the values in the list 'x'
```
Copy

Output —> 22.28175

Now we can try to extract specific values from the list 'y' to find the standard deviation.

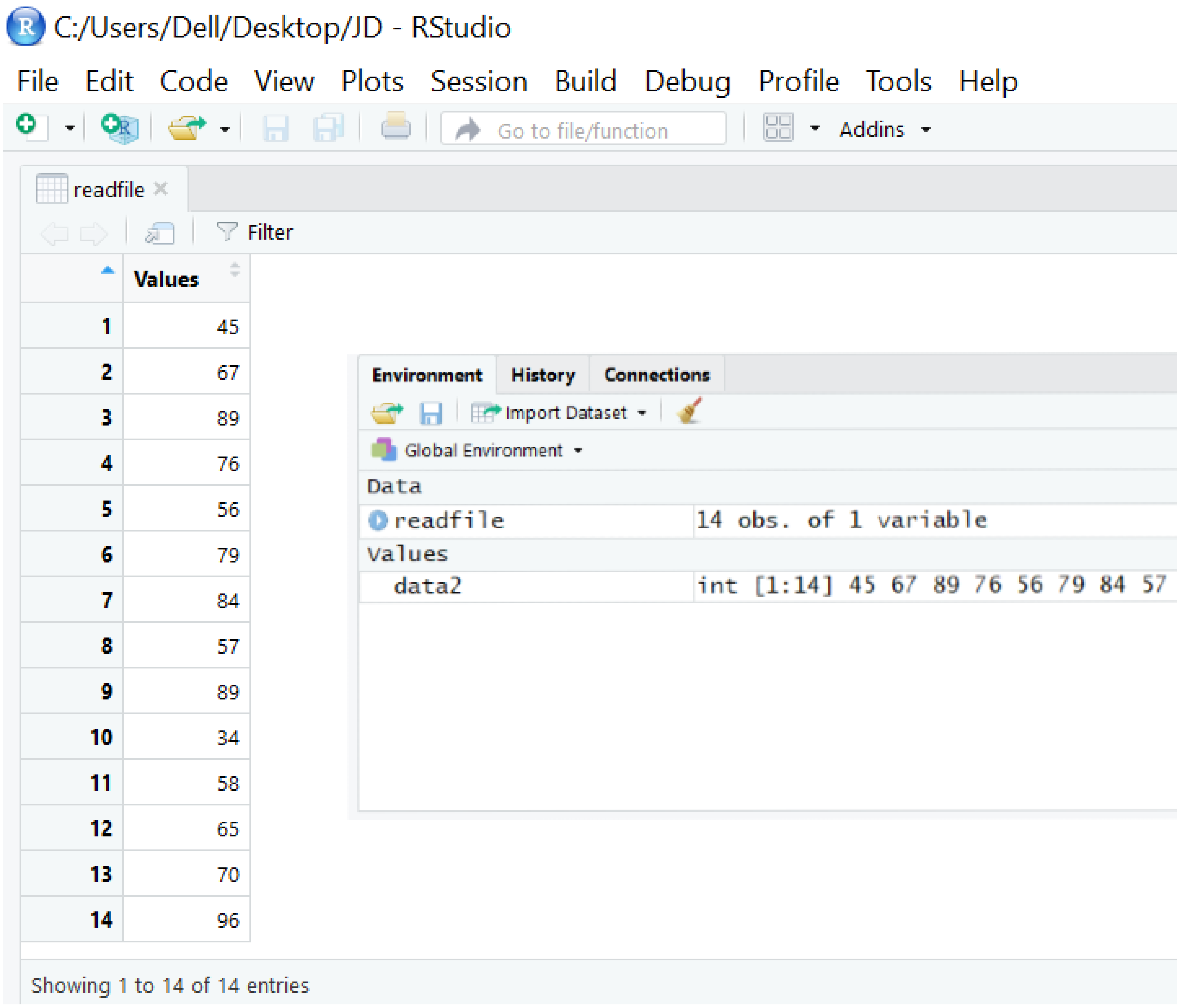
```
 y <- c(34,65,78,96,56,78,54,57,89)  #creates a list 'y' having some
values

data1 <- y[1:5] #extract specific values using its Index

sd(data1) #calculates the standard deviation for Indexed or extracted
values from the list.
```
Copy

Output —> 23.28519

Finding the Standard deviation of the values stored in a CSV file

In this method, we are importing a CSV file to find the standard deviation in R for the values which are stored in that file.

```
readfile <- read.csv('testdata1.csv')  #reading a csv file

data2 <- readfile$Values      #getting values stored in the header
'Values'

sd(data2)                              #calculates the standard deviation
```
Copy


C:/Users/Dell/Desktop/JD - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
readfile
Filter
Environment History Connections
Import Dataset
Global Environment

	Values
1	45
2	67
3	89
4	76
5	56
6	79
7	84
8	57
9	89
10	34
11	58
12	65
13	70
14	96

Showing 1 to 14 of 14 entries

```
Data
readfile          14 obs. of 1 variable
Values
  data2           int [1:14] 45 67 89 76 56 79 84 57
```

Output —> 17.88624

High and Low Standard Deviation

In general, The values will be so close to the average value in **low standard deviation** and the values will be far spread from the average value in the **high standard deviation.**

We can illustrate this with an example.

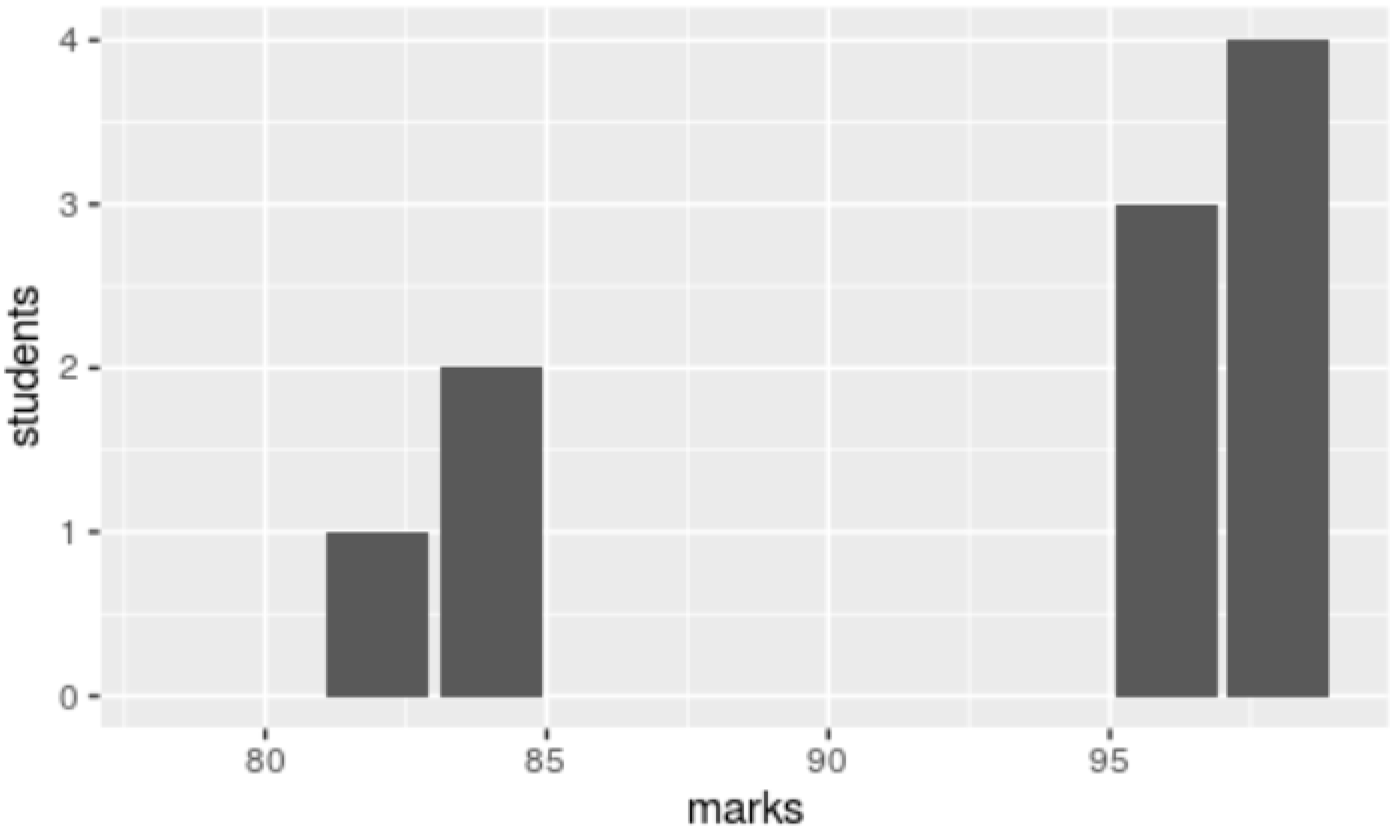
```
x <- c(79,82,84,96,98)
mean(x)
--->  82.22222
sd(x)
--->  10.58038
```
Copy

To plot these values in a bar graph using in R, run the below code.

To install the ggplot2 package, run this code in R studio.

–-> install.packages("ggplot2")

```
library(ggplot2)

values <- data.frame(marks=c(79,82,84,96,98), students=c(0,1,2,3,4,))
head(values)                    #displayes the values
 marks students
1    79         0
2    82         1
3    84         2
4    96         3
5    98         4
x <- ggplot(values, aes(x=marks, y=students))+geom_bar(stat='identity')
x                               #displays the plot
```
Copy



In the above results, you can observe that most of the data is clustering around the mean value(79,82,84) which shows that it is a **low standard deviation**.

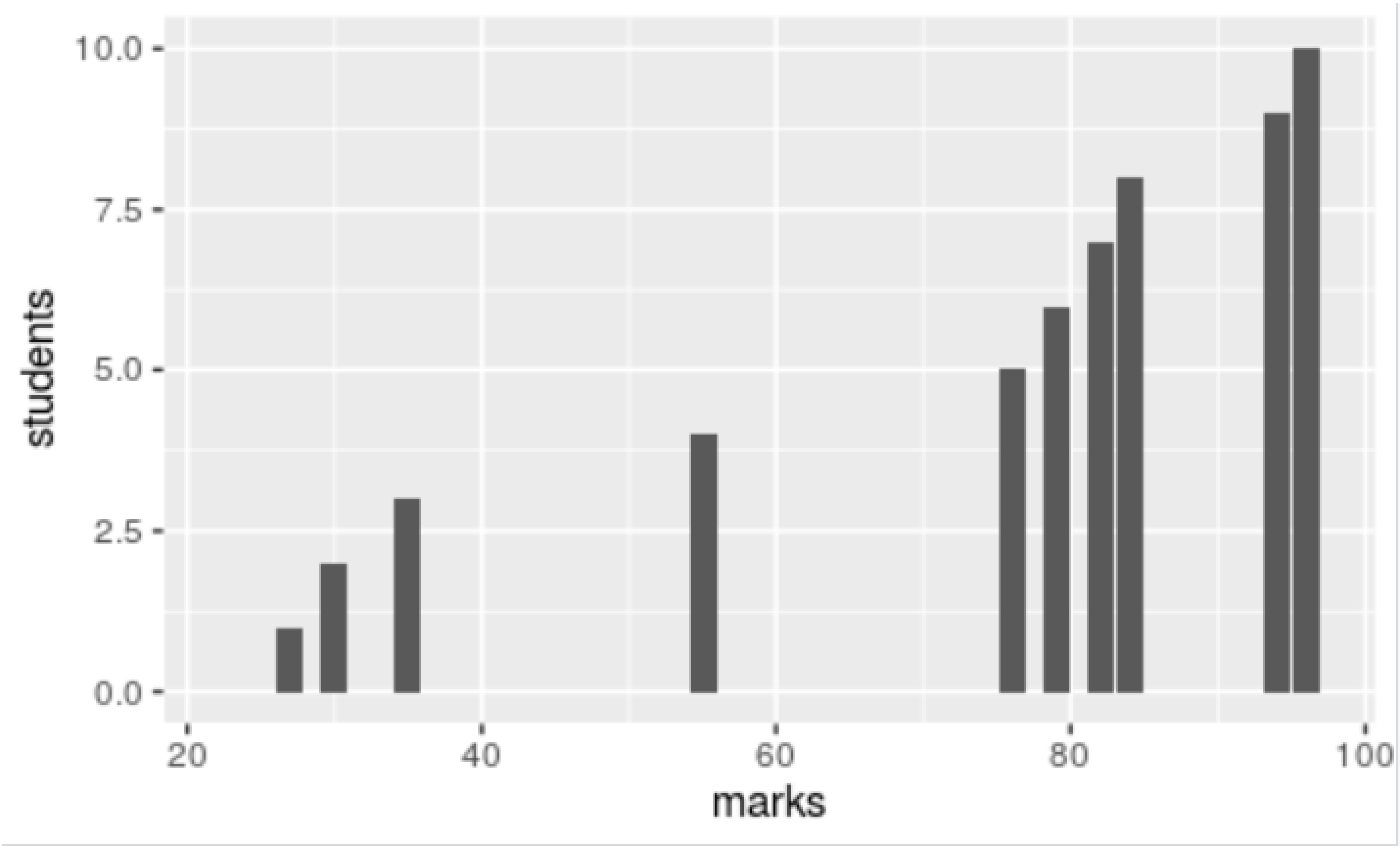
Illustration for **high standard deviation**.

```
y <- c(23,27,30,35,55,76,79,82,84,94,96)
mean(y)
---> 61.90909
sd(y)
---> 28.45507
```
Copy

To plot these values using a bar graph in ggplot in R, run the below code.

```
library(ggplot2)

values <- data.frame(marks=c(23,27,30,35,55,76,79,82,84,94,96),
students=c(0,1,2,3,4,5,6,7,8,9,10))
head(values)                        #displayes the values
  marks students
```

```
1     23         0
2     27         1
3     30         2
4     35         3
5     55         4
6     76         5
x <- ggplot(values, aes(x=marks, y=students))+geom_bar(stat='identity')
x                               #displays the plot
```
Copy



In the above results, you can see the widespread data. You can see the least score of 23 which is very far from the average score 61. This is called the **high standard deviation**

By now, you got a fair understanding of using the sd(' ') function to calculate the standard deviation in the R language. Let's sum up this tutorial by solving simple problems.

Example #1: Standard Deviation for a List of Even Numbers

Find the standard deviation of the even numbers between 1-20 (exclude 1 and 20).

Solution: The even numbers between 1 to 20 are,

–-> 2, 4, 6, 8, 10, 12, 14, 16, 18

Lets find the standard deviation of these values.

```
x <- c(2,4,6,8,10,12,14,16,18)  #list of even numbers from 1 to 20

sd(x)                           #calculates the standard deviation of
these
                            values in the list of even numbers from 1 to
20
```
Copy

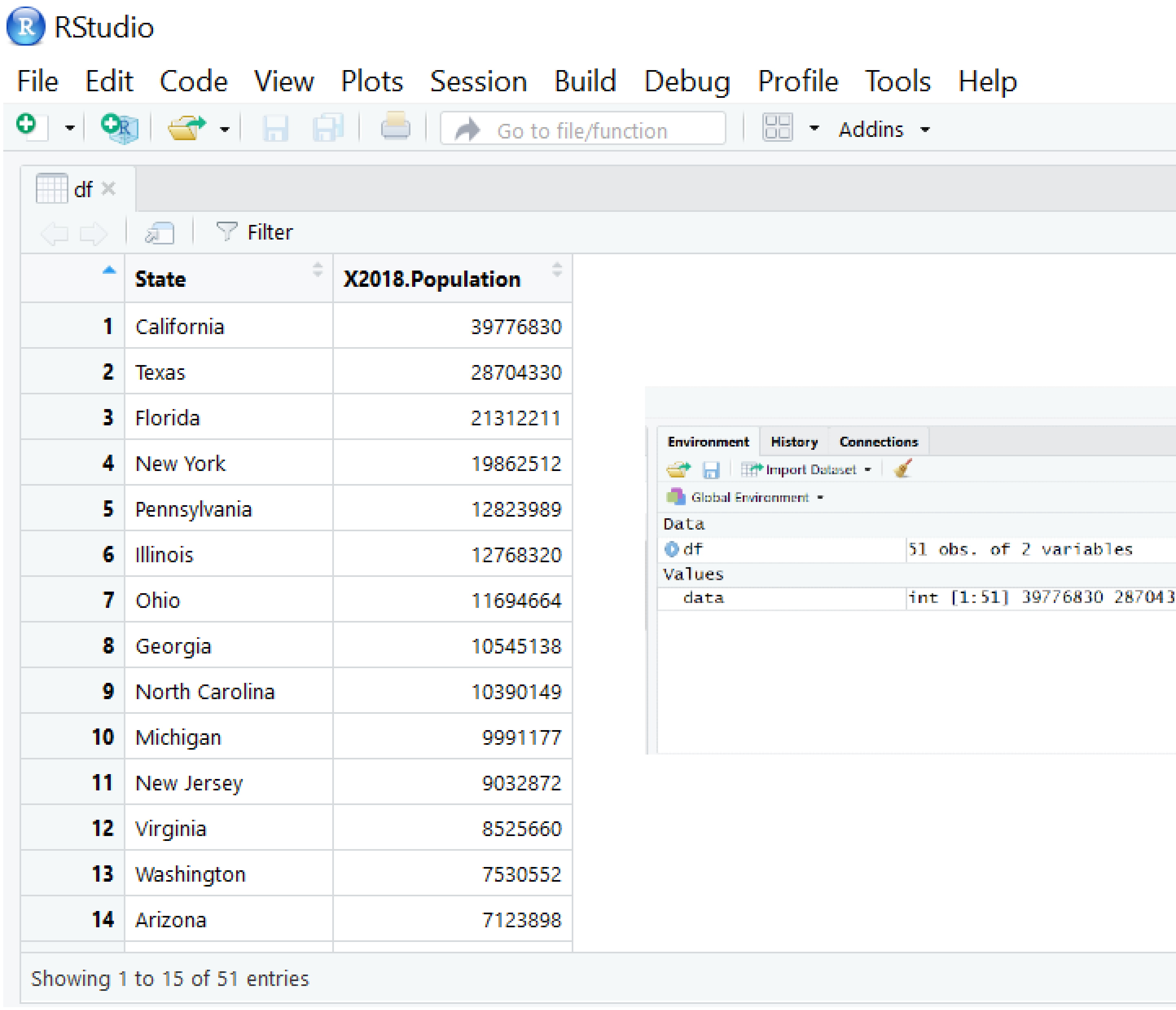
Output —> 5.477226

Example #2: Standard Deviation for US Population Data

Find the standard deviation of the state-wise population in the USA.

For this, import the CSV file and read the values to find the standard deviation and plot the result in a histogram in R.

```
df<-read.csv("population.csv")      #reads csv file
data<-df$X2018.Population           #extarcts the data from population
                                     column
mean(data)                          #calculates the mean

View(df)                            #displays the data
sd(data)                            #calculates the standard deviation
```
Copy


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
df
Filter
Environment History Connections
Import Dataset
Global Environment

	State	X2018.Population
1	California	39776830
2	Texas	28704330
3	Florida	21312211
4	New York	19862512
5	Pennsylvania	12823989
6	Illinois	12768320
7	Ohio	11694664
8	Georgia	10545138
9	North Carolina	10390149
10	Michigan	9991177
11	New Jersey	9032872
12	Virginia	8525660
13	Washington	7530552
14	Arizona	7123898

Showing 1 to 15 of 51 entries

```
Data
df          51 obs. of 2 variables
Values
  data      int [1:51] 39776830 287043
```

Output ----> **mean** = 6432008, **Sd** = 7376752

How to find the mean of all values in an R data frame?

we are going to find the mean of the values of a dataframe in R with the use of mean() function.

Syntax:
mean(dataframe)

Creating a Dataframe
A dataframe can be created with the use of data.frame() function that is pre-defined in the R library. This function accepts the elements and the number of rows and columns that are required for the dataframe to be created.

```
# R Program to create a dataframe

 # Creating a Data Frame

df<-data.frame(row1 = 0:2, row2 = 3:5, row3 = 6:8)

print(df)
```

Output:

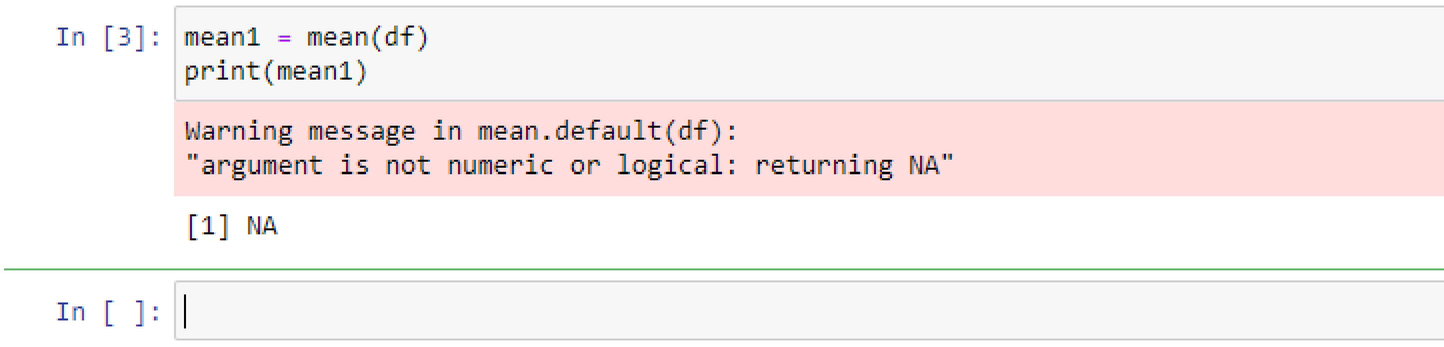
```
  row1 row2 row3

1    0    3    6

2    1    4    7

3    2    5    8
```

Computing Mean of the Dataframe
R language provides an in-built function mean() to compute the mean of a dataframe. Following is an R program for the implementation of mean().

- R

```
mean1 = mean(df)

print(mean1)
```

Output:



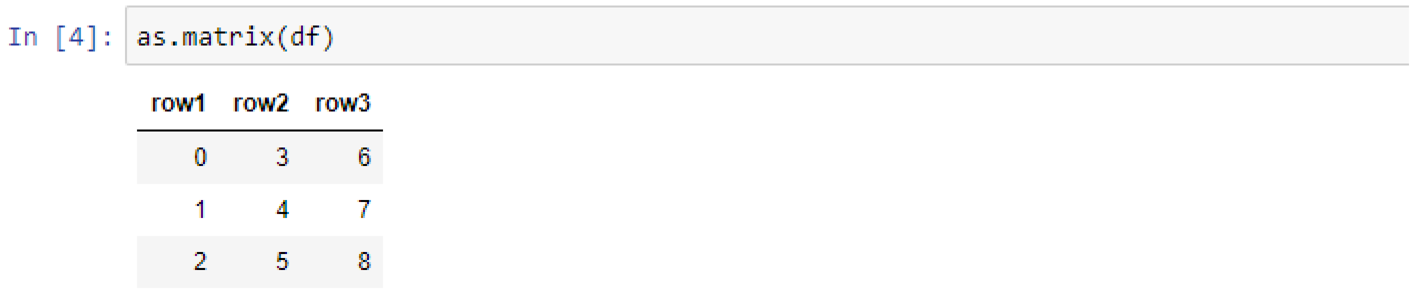
```
In [3]: mean1 = mean(df)
        print(mean1)

        Warning message in mean.default(df):
        "argument is not numeric or logical: returning NA"

        [1] NA

In [ ]:
```

Here in the above code, a warning message is displayed, which returns NA because the dataframe is not in a numeric form. There is a need to convert it into matrix form to compute the mean of the dataframe. R provides an inbuilt as.matrix() function to convert a dataframe to a matrix.

- R

```
# Converting dataframe to matrixa

as.matrix(df)
```

Output:



```
In [4]: as.matrix(df)
```

row1	row2	row3
0	3	6
1	4	7
2	5	8

Now, to compute mean of this matrix created from a dataframe, use the mean function on the matrix.

- R

```
# Finding mean of the dataframe



# Using mean() function

mean(as.matrix(df))
```

Output:

4

Example 2:

- R

```
# R program to illustrate dataframe

Roll_num = c(01, 02, 03)

Age = c(22, 25, 45)

Marks = c(70, 80, 90)



# To create dataframe use data.frame command and

# then pass each of the vectors

# we have created as arguments

# to the function data.frame()
```

```
df = data.frame(Roll_num, Age, Marks)



print(df)
```

Output:

```
      Roll_num Age Marks
1            1  22    70
2            2  25    80
3            3  45    90
```

- R

```
# Computing mean of the above dataframe



# Using the mean() function

mean(as.matrix(df))
```

Output:

```
37.5555555555556
```

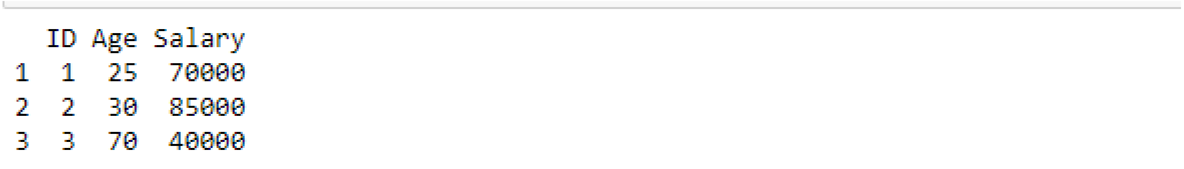
Example 3:

- R

```
# R program to illustrate dataframe

ID = c(01, 02, 03)

Age = c(25, 30, 70)
```

```
Salary = c(70000, 85000, 40000)



# To create dataframe use data.frame command and

# then pass each of the vectors

# we have created as arguments

# to the function data.frame()

df = data.frame(ID, Age, Salary)



print(df)
```

Output:



```
  ID Age Salary
1  1  25  70000
2  2  30  85000
3  3  70  40000
```

- R

```
# Computing mean of the dataframe



# Using mean() function

mean(as.matrix(df))
```

Output:

```
21681.2222222222
```

Calculate the Median in R

How to calculate the median of a DataFrame column or a Vector in R? The median() is a base function in R that is used to calculate the median of a Vector. The median of a dataset is the value that, assuming the dataset is ordered from smallest to largest, falls in the middle. If there are an even number of values in a dataset, the middle two values are the median.

This function accepts a vector as input and returns the median as a numeric value.

1. Syntax of median()

The following is the syntax of the median() function that calculates the median value.

```
# Syntax of median
median(x, na.rm = FALSE, …)
```

Copy

Parameters:

- **x** – It is an input vector of type Numeric
- **na.rm** – Defaults to FALSE. When TRUE, it ignores NA value.

2. R Median of DataFrame Column

By using R base function median() let's calculate the median value of the DataFrame column. The following example demonstrates getting median with and with out NA values on a column.

```
# Create Data Frame
df <- data.frame(id=c(11,25,50,42,55),
              price=c(144,NA,321,567,567))
df

# Calculate median of DataFrame column
res <- median(df$id)
res
```

Copy

Yields below output.

Calculating the median on a column that has `NA` values results in NA, you need to ignore the NA to get the right result. Let's calculate the median on the column that has NA values by using the `na.rm` param to ignore NA values. On our DataFrame, we have a column `price` that has NA values.

```
# with NA
res <- median(df$price, na.rm=TRUE)
res

# Output
# [1] 444
```
Copy

3. R Median of Vector

Similarly, let's also calculate the median from the values of Vector. The following examples demonstrate calculating the median when you have an even count and odd count of vector and also when you have NA values.

```
# Calculate median of Vector
vec = c(6,7,8)
median(vec)

# Output
# [1] 7

# Calculate mean of Vector
vec = c(6,7,8, 9)
median(vec, na.rm=TRUE)
```

```
# Output
# [1] 7.5

# Calculate mean of Vector
vec = c(10,11,6,7,8,9, NA)
median(vec, na.rm=TRUE)

# Output
# [1] 8.5
```

How to Find the Range in R using the Range () function (with Examples)

- Home
- Functional R – An R Function Tutorial
- How to Find the Range in R using the Range () function (with Examples)

R has an efficient way to get the minimum and maximum values within a vector: the range() function. The range is the interval or difference between the lowest and the highest value within the data vector. This has numerous practical uses when analyzing a data frame and is required by many statistical control and graph functions (if you are writing packages or helper utilities). As a general rule, range based measurements (effectively a limit estimate) are easier to explain to laymen than a standard deviation or other common summary statistics.

The range is also required when building and validating statistical process control models, since you will want to ensure the model is accurate and relevant for the full range, interval, or difference of possible values in the dataset or dataframe.

Finding the Minimum Value and Maximum Value

Within R, the range function uses the following syntax: range (vector)

The implementation of this R chart function is extremely efficient and runs within Base R. That being said, the range function requires iteration across the full numeric vector when called, so incorporate this consideration into your design.

The Range Function in R

In R, the range function has the format of range(vector) and it produces the smallest and largest values of the numeric vector that is being evaluated. The result is that you have the range chart of values covered by the data set.

```
# range in R
> x=c(5,2,7,9,4)
 > range(x)
 [1] 2 9
```

If you examine the example, you will see the results accurately captured the range of the vector: 2 for the minimum, 9 for the maximum.

Range in R – Missing values option

When dealing with the NA value range in R has a logical option in the form of na.rm. The na.rm parameter, which means NA remove, can be TRUE or FALSE. This helps protect you from missing value errors. If the logical option is FALSE, which it is by default if omitted, the function returns an NA value for both the minimum value and maximum value. If it is TRUE, then, NA values are discounted.

```
# range in R - the NA issue
> x=c(5,2,NA,9,4)
 > range(x,na.rm=FALSE)
 [1] NA NA
```

Here, we have the case where na.rm is FALSE. Note that both resulting values are NA, this indicates that there is no answer.

```
# range in r - using na.rm to clean up results
> range(x,na.rm=TRUE)
 [1] 2 9
```

Here, na.rm is TRUE and the NA value is ignored resulting in a minimum and maximum values.

Range in R – Character data

Here, we have the case of applying the range function to a character vector. When applied to characters, a range analysis returns the first and last of the characters in alphabetical order- the minimum value is the first character vector on the list, and the maximum value is the last variable on the list, alphabetically.

```
# range in R - vectors with alphabetical data
> x=c("c","r","e","a","g","e","r")
> range(x)
```

```
[1] "a" "r"
```

In this example, there are 2 e's and 2 r's, however, because duplications of multiple variables do not matter it has no effect on the summary statistics.

Finding the range of a data set or dataframe effectively gives you its boundaries. The range () function in R provides you with the visible extreme number at the max and min of the series. Note that these may not necessarily represent the real world limits of a series if your sample size or subgroup size is small and / or extreme values are very rare. Most lottery ticket samples have payouts ranging between $0 and $100; a tiny handful of potential samples have payouts massively above that….

The apply family

R offers a family of **apply functions**, which allow you to apply a function across different chunks of data. Offers an alternative to explicit iteration using `for()` loop; can be simpler and faster, though not always. Summary of functions:

- `apply()`: apply a function to rows or columns of a matrix or data frame
- `lapply()`: apply a function to elements of a list or vector
- `sapply()`: same as the above, but simplify the output (if possible)
- `tapply()`: apply a function to levels of a factor vector

`apply()`: rows or columns of a matrix or data frame

The `apply()` function takes inputs of the following form:

- `apply(x, MARGIN=1, FUN=my.fun)`, to apply `my.fun()` across rows of a matrix or data frame `x`
- `apply(x, MARGIN=2, FUN=my.fun)`, to apply `my.fun()` across columns of a matrix or data frame `x`

```
apply(state.x77, MARGIN=2, FUN=min) # Minimum entry in each column
## Population     Income Illiteracy   Life Exp     Murder    HS Grad       Fro
st
##     365.00    3098.00       0.50      67.96       1.40      37.80        0.
00
##       Area
##    1049.00
apply(state.x77, MARGIN=2, FUN=max) # Maximum entry in each column
```

```
## Population     Income Illiteracy   Life Exp     Murder    HS Grad       Fro
st
##    21198.0     6315.0        2.8       73.6       15.1       67.3       188
.0
##       Area
##   566432.0
apply(state.x77, MARGIN=2, FUN=which.max) # Index of the max in each column
## Population     Income Illiteracy   Life Exp     Murder    HS Grad       Fro
st
##          5          2         18         11          1         44
28
##       Area
##          2
apply(state.x77, MARGIN=2, FUN=summary) # Summary of each col, get back matri
x!
##           Population  Income Illiteracy Life Exp Murder HS Grad  Frost
Area
## Min.         365.00 3098.00      0.500  67.9600  1.400  37.800   0.00   104
9.00
## 1st Qu.     1079.50 3992.75      0.625  70.1175  4.350  48.050  66.25  3698
5.25
## Median      2838.50 4519.00      0.950  70.6750  6.850  53.250 114.50  5427
7.00
## Mean        4246.42 4435.80      1.170  70.8786  7.378  53.108 104.46  7073
5.88
## 3rd Qu.     4968.50 4813.50      1.575  71.8925 10.675  59.150 139.75  8116
2.50
## Max.       21198.00 6315.00      2.800  73.6000 15.100  67.300 188.00 56643
2.00
```

Applying a custom function

For a custom function, we can just define it before hand, and the use `apply()` as usual

```
# Our custom function: trimmed mean
trimmed.mean = function(v) {
  q1 = quantile(v, prob=0.1)
  q2 = quantile(v, prob=0.9)
  return(mean(v[q1 <= v & v <= q2]))
}
```

```
apply(state.x77, MARGIN=2, FUN=trimmed.mean)
## Population      Income  Illiteracy    Life Exp      Murder     HS Grad
## 3384.27500  4430.07500     1.07381    70.91775     7.29750    53.33750
##      Frost        Area
##  104.68293 56575.72500
```

We'll learn more about functions later (don't worry too much at this point about the details of the function definition)

Applying a custom function "on-the-fly"

Instead of defining a custom function before hand, we can just define it "on-the-fly". Sometimes this is more convenient

```
# Compute trimmed means, defining this on-the-fly
apply(state.x77, MARGIN=2, FUN=function(v) {
  q1 = quantile(v, prob=0.1)
  q2 = quantile(v, prob=0.9)
  return(mean(v[q1 <= v & v <= q2]))
})
## Population      Income  Illiteracy    Life Exp      Murder     HS Grad
## 3384.27500  4430.07500     1.07381    70.91775     7.29750    53.33750
##      Frost        Area
##  104.68293 56575.72500
```

Applying a function that takes extra arguments

Can tell `apply()` to pass **extra arguments** to the function in question. E.g., can use: `apply(x, MARGIN=1, FUN=my.fun, extra.arg.1, extra.arg.2)`, for two extra arguments `extra.arg.1`, `extra.arg.2` to be passed to `my.fun()`

```
# Our custom function: trimmed mean, with user-specified percentiles
trimmed.mean = function(v, p1, p2) {
  q1 = quantile(v, prob=p1)
  q2 = quantile(v, prob=p2)
  return(mean(v[q1 <= v & v <= q2]))
}


apply(state.x77, MARGIN=2, FUN=trimmed.mean, p1=0.01, p2=0.99)
```

```
##    Population        Income    Illiteracy      Life Exp        Murder       HS G
rad
##   3974.125000   4424.520833      1.136735     70.882708      7.341667     53.131
250
##         Frost          Area
##    104.895833 61860.687500
```

What’s the return argument?

What kind of data type will `apply()` give us? Depends on what function we pass. Summary, say, with `FUN=my.fun()`:

- If `my.fun()` returns a single value, then `apply()` will return a vector
- If `my.fun()` returns k values, then `apply()` will return a matrix with k rows (note: this is true *regardless* of whether `MARGIN=1` or `MARGIN=2`)
- If `my.fun()` returns different length outputs for different inputs, then `apply()` will return a list
- If `my.fun()` returns a list, then `apply()` will return a list

We’ll grapple with this on the lab. This is one main advantage of `purrr` package: there is a much more transparent return object type

Optimized functions for special tasks

Don’t overuse the apply paradigm! There’s lots of special functions that **optimized** are will be both simpler and faster than using `apply()`. E.g.,

- `rowSums()`, `colSums()`: for computing row, column sums of a matrix
- `rowMeans()`, `colMeans()`: for computing row, column means of a matrix
- `max.col()`: for finding the maximum position in each row of a matrix

Combining these functions with logical indexing and vectorized operations will enable you to do quite a lot. E.g., how to count the number of positives in each row of a matrix?

```
x = matrix(rnorm(9), 3, 3)
# Don't do this (much slower for big matrices)
apply(x, MARGIN=1, function(v) { return(sum(v > 0)) })
```

```
## [1] 1 3 0
```

```
# Do this insted (much faster, simpler)
rowSums(x > 0)
```

```
## [1] 1 3 0
```

Part III

`lapply()`, `sapply()`, `tapply()`

`lapply()`: elements of a list or vector

The `lapply()` function takes inputs as in: `lapply(x, FUN=my.fun)`, to apply `my.fun()` across elements of a list or vector `x`. The output is always a list

```
my.list
```

```
## $nums
## [1] 0.1 0.2 0.3 0.4 0.5 0.6
##
## $chars
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
##
## $bools
## [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE
```

```
lapply(my.list, FUN=mean) # Get a warning: mean() can't be applied to chars
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
## $nums
## [1] 0.35
##
## $chars
## [1] NA
##
## $bools
## [1] 0.6666667
```

```
lapply(my.list, FUN=summary)
```

```
## $nums
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.100   0.225   0.350   0.350   0.475   0.600
##
```

```
## $chars
##    Length     Class      Mode
##        12 character character
##
## $bools
##    Mode   FALSE    TRUE
## logical       2       4
```

`sapply()`: elements of a list or vector

The `sapply()` function works just like `lapply()`, but tries to **simplify** the return value whenever possible. E.g., most common is the conversion from a list to a vector

```
sapply(my.list, FUN=mean) # Simplifies the result, now a vector
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
##      nums     chars     bools
## 0.3500000        NA 0.6666667
sapply(my.list, FUN=summary) # Can't simplify, so still a list
## $nums
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.100   0.225   0.350   0.350   0.475   0.600
##
## $chars
##    Length     Class      Mode
##        12 character character
##
## $bools
##    Mode   FALSE    TRUE
## logical       2       4
```

`tapply()`: levels of a factor vector

The function `tapply()` takes inputs as in: `tapply(x, INDEX=my.index, FUN=my.fun)`, to apply `my.fun()` to subsets of entries in `x` that share a common level in `my.index`

```
# Compute the mean and sd of the Frost variable, within each region
tapply(state.x77[,"Frost"], INDEX=state.region, FUN=mean)
##     Northeast         South North Central          West
##      132.7778       64.6250      138.8333      102.1538
tapply(state.x77[,"Frost"], INDEX=state.region, FUN=sd)
##     Northeast         South North Central          West
##      30.89408      31.30682      23.89307      68.87652
```

`split()`: split by levels of a factor

The function `split()` split up the rows of a data frame by levels of a factor, as in: `split(x, f=my.index)` to split a data frame `x` according to levels of `my.index`

```
# Split up the state.x77 matrix according to region
state.by.reg = split(data.frame(state.x77), f=state.region)
class(state.by.reg) # The result is a list
## [1] "list"
names(state.by.reg) # This has 4 elements for the 4 regions
## [1] "Northeast"     "South"         "North Central" "West"
class(state.by.reg[[1]]) # Each element is a data frame
## [1] "data.frame"
```

```
# For each region, display the first 3 rows of the data frame
lapply(state.by.reg, FUN=head, 3)
## $Northeast
##                 Population Income Illiteracy Life.Exp Murder HS.Grad Frost
Area
## Connecticut           3100   5348        1.1    72.48    3.1    56.0   139
4862
## Maine                 1058   3694        0.7    70.39    2.7    54.7   161 3
0920
## Massachusetts         5814   4755        1.1    71.83    3.3    58.5   103
7826
##
## $South
##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area
## Alabama         3615   3624        2.1    69.05   15.1    41.3    20 50708
## Arkansas        2110   3378        1.9    70.66   10.1    39.9    65 51945
## Delaware         579   4809        0.9    70.06    6.2    54.6   103  1982
##
## $`North Central`
##           Population Income Illiteracy Life.Exp Murder HS.Grad Frost  Area
## Illinois       11197   5107        0.9    70.14   10.3    52.6   127 55748
## Indiana         5313   4458        0.7    70.88    7.1    52.9   122 36097
## Iowa            2861   4628        0.5    72.56    2.3    59.0   140 55941
##
## $West
##             Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Ar
ea
## Alaska             365   6315        1.5    69.31   11.3    66.7   152 5664
32
## Arizona           2212   4530        1.8    70.55    7.8    58.1    15 1134
17
## California       21198   5114        1.1    71.71   10.3    62.6    20 1563
61
```

```r
# For each region, average each of the 8 numeric variables
lapply(state.by.reg, FUN=function(df) {
  return(apply(df, MARGIN=2, mean))
})
```

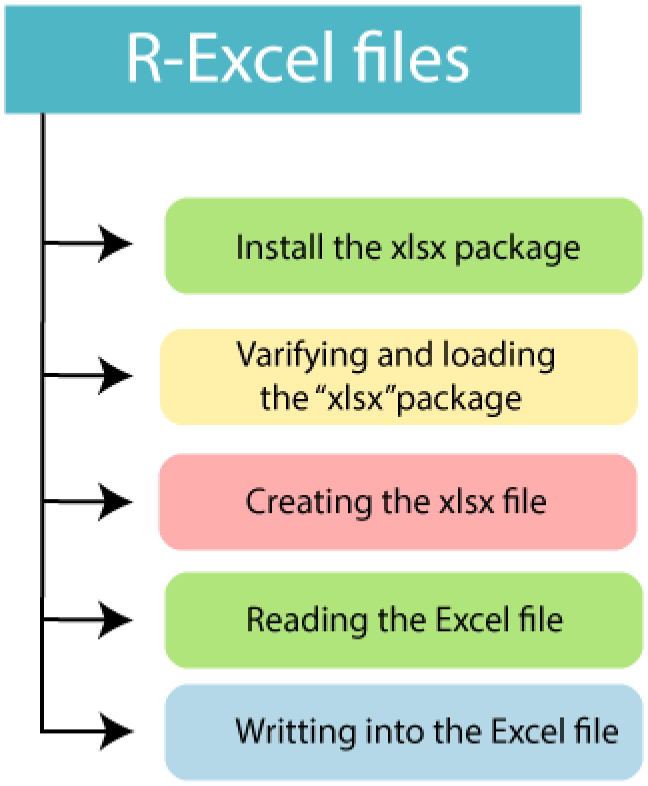
```
## $Northeast
##   Population        Income   Illiteracy      Life.Exp        Murder       HS.G
rad
##  5495.111111  4570.222222     1.000000     71.264444      4.722222     53.966
667
##        Frost          Area
##   132.777778 18141.000000
##
## $South
##  Population       Income  Illiteracy     Life.Exp       Murder      HS.Grad
##  4208.12500  4011.93750     1.73750    69.70625    10.58125    44.34375
##       Frost        Area
##    64.62500 54605.12500
##
## $`North Central`
##  Population      Income  Illiteracy    Life.Exp      Murder     HS.Grad
##  4803.00000  4611.08333     0.70000    71.76667     5.27500    54.51667
##       Frost        Area
##   138.83333 62652.00000
##
## $West
##   Population       Income   Illiteracy     Life.Exp       Murder      HS.G
rad
## 2.915308e+03 4.702615e+03 1.023077e+00 7.123462e+01 7.215385e+00 6.200000e
+01
##        Frost         Area
## 1.021538e+02 1.344630e+05
```

Summary

- Data frames are a representation of the "classic" data table in R: rows are observations/cases, columns are variables/features
- Each column can be a different data type (but must be the same length)
- `subset()`: function for extracting rows of a data frame meeting a condition
- `split()`: function for splitting up rows of a data frame, according to a factor variable
- `apply()`: function for applying a given routine to rows or columns of a matrix or data frame
- `lapply()`: similar, but used for applying a routine to elements of a vector or list
- `sapply()`: similar, but will try to simplify the return type, in comparison to `lapply()`
- `tapply()`: function for applying a given routine to groups of elements in a vector or list, according to a factor variable

R Excel file

The xlsx is a file extension of a spreadsheet file format which was created by Microsoft to work with Microsoft Excel. In the present era, Microsoft Excel is a widely used spreadsheet program that sores data in the .xls or .xlsx format. R allows us to read data directly from these files by providing some excel specific packages. There are lots of packages such as XLConnect, xlsx, gdata, etc. We will use xlsx package, which not only allows us to read data from an excel file but also allow us to write data in it.



Install xlsx Package

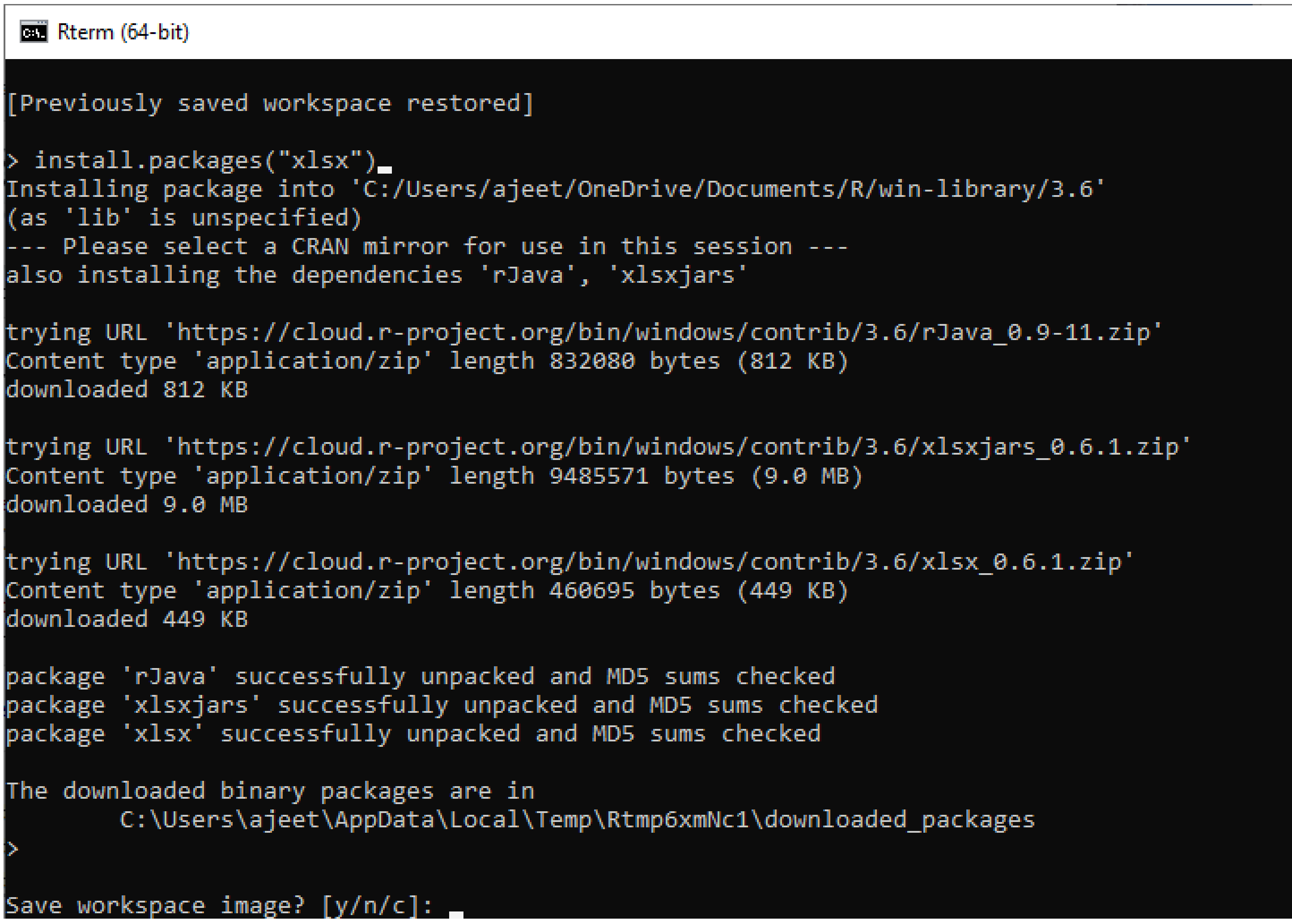
Our primary task is to install "xlsx" package with the help of install.package command. When we install the xlsx package, it will ask us to install some additional packages on which this package is dependent. For installing the additional packages, the same command is used with the required package name. There is the following syntax of install command:

```
install.packages("package name")
```

Example

```
install.packages("xlsx")
```

Output



Rterm (64-bit)

```
[Previously saved workspace restored]

> install.packages("xlsx")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
also installing the dependencies 'rJava', 'xlsxjars'

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/rJava_0.9-11.zip'
Content type 'application/zip' length 832080 bytes (812 KB)
downloaded 812 KB

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsxjars_0.6.1.zip'
Content type 'application/zip' length 9485571 bytes (9.0 MB)
downloaded 9.0 MB

trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsx_0.6.1.zip'
Content type 'application/zip' length 460695 bytes (449 KB)
downloaded 449 KB

package 'rJava' successfully unpacked and MD5 sums checked
package 'xlsxjars' successfully unpacked and MD5 sums checked
package 'xlsx' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\ajeet\AppData\Local\Temp\Rtmp6xmNc1\downloaded_packages
>

Save workspace image? [y/n/c]:
```

Verifying and Loading of "xlsx" Package

In R, grepl() and any() functions are used to verify the package. If the packages are installed, these functions will return True else return False. For verifying the package, both the functions are used together.

For loading purposes, we use the library() function with the appropriate package name. This function loads all the additional packages also.

Example

```
#Installing xlsx package
install.packages("xlsx")
```

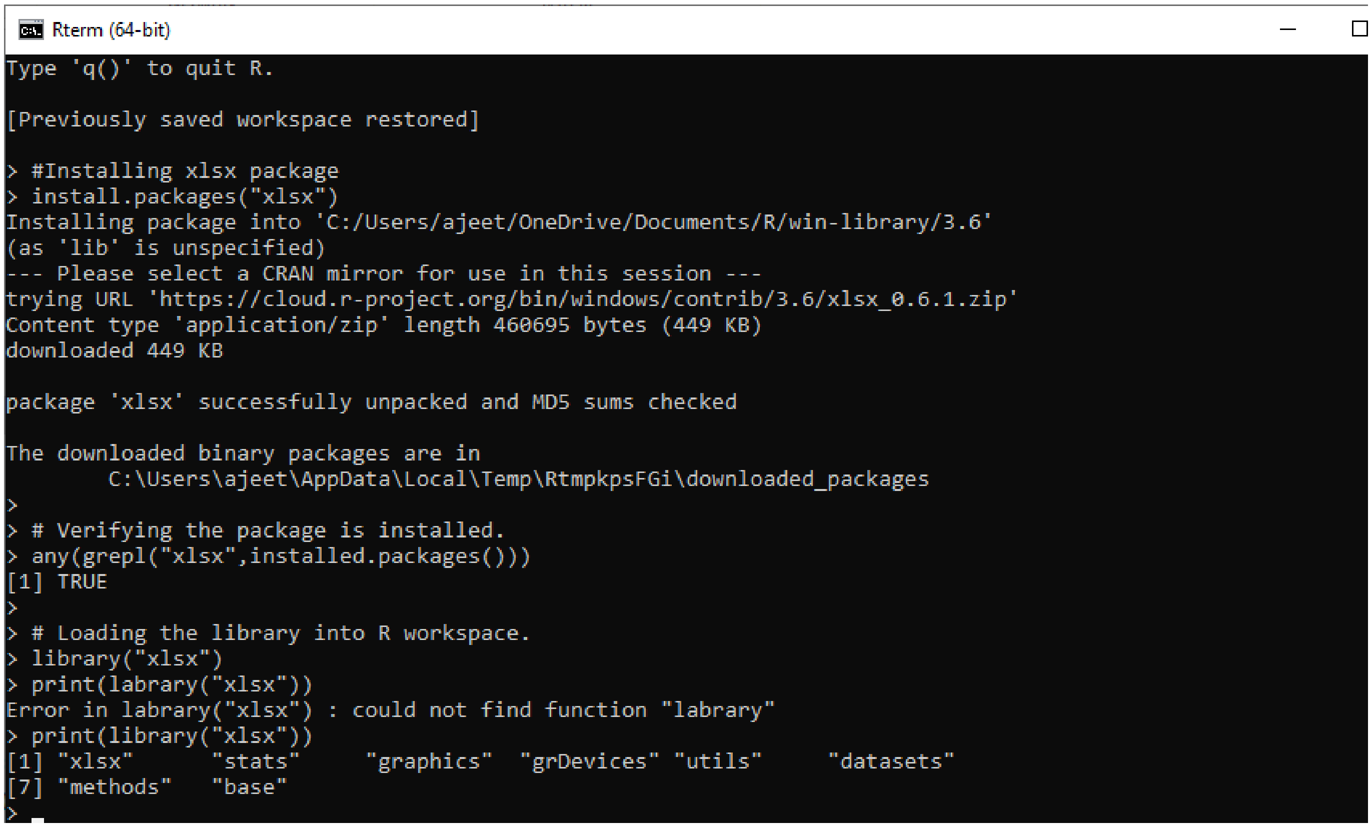
Verifying the package is installed.

any(grepl("xlsx",installed.packages()))

Loading the library into R workspace.

library("xlsx")

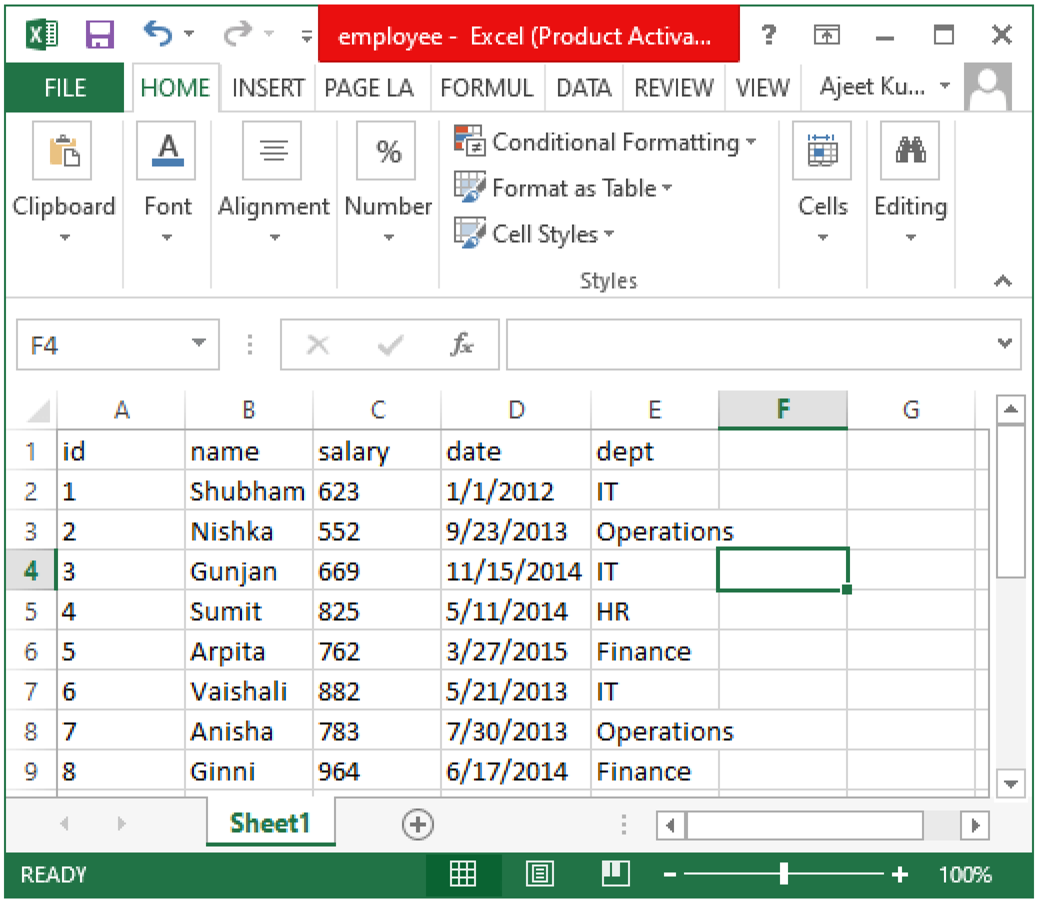
Output



```
Type 'q()' to quit R.

[Previously saved workspace restored]

> #Installing xlsx package
> install.packages("xlsx")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsx_0.6.1.zip'
Content type 'application/zip' length 460695 bytes (449 KB)
downloaded 449 KB

package 'xlsx' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\ajeet\AppData\Local\Temp\RtmpkpsFGi\downloaded_packages
>
> # Verifying the package is installed.
> any(grepl("xlsx",installed.packages()))
[1] TRUE
>
> # Loading the library into R workspace.
> library("xlsx")
> print(labrary("xlsx"))
Error in labrary("xlsx") : could not find function "labrary"
> print(library("xlsx"))
[1] "xlsx"      "stats"     "graphics"  "grDevices" "utils"     "datasets"
[7] "methods"   "base"
>
```

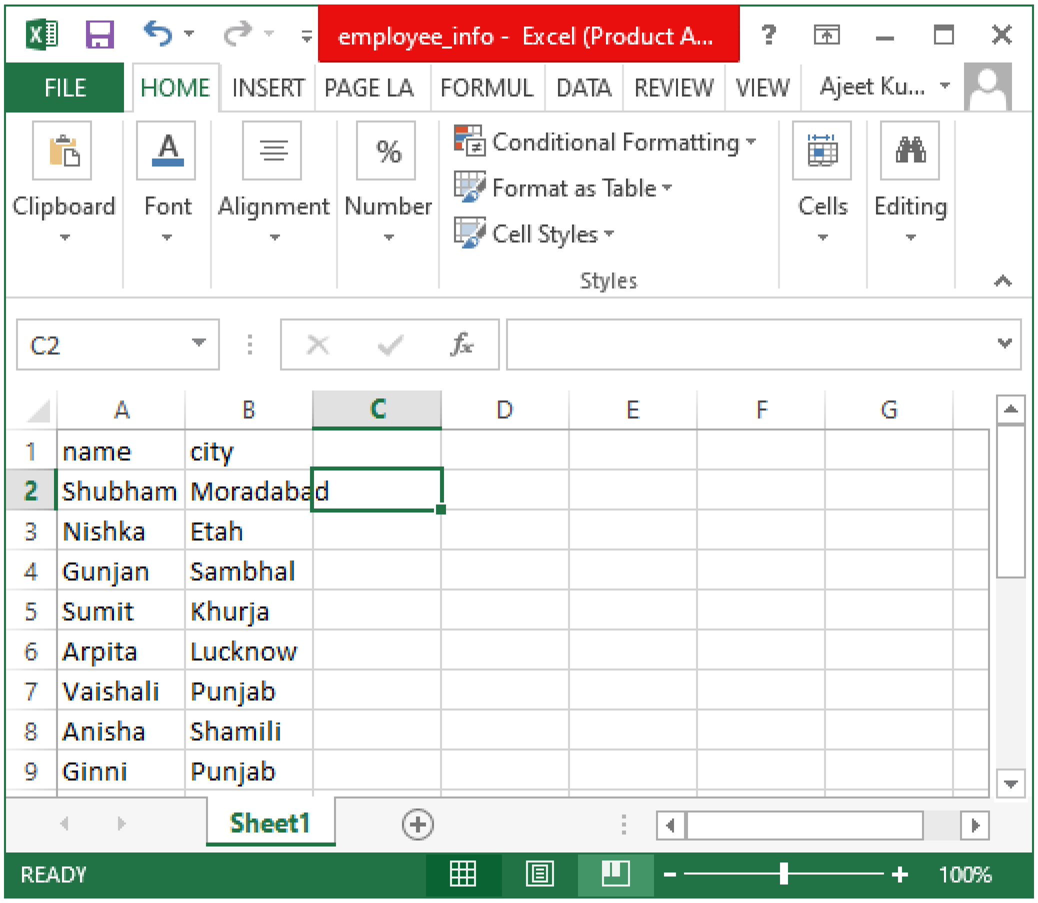
Creating an xlsx File

Once the xlsx package is loaded into our system, we will create an excel file with the following data and named it employee.



	A	B	C	D	E	F	G
1	id	name	salary	date	dept		
2	1	Shubham	623	1/1/2012	IT		
3	2	Nishka	552	9/23/2013	Operations		
4	3	Gunjan	669	11/15/2014	IT		
5	4	Sumit	825	5/11/2014	HR		
6	5	Arpita	762	3/27/2015	Finance		
7	6	Vaishali	882	5/21/2013	IT		
8	7	Anisha	783	7/30/2013	Operations		
9	8	Ginni	964	6/17/2014	Finance		

Apart from this, we will create another table with the following data and give it a name as employee_info.



	A	B	C	D	E	F	G
1	name	city					
2	Shubham	Moradabad					
3	Nishka	Etah					
4	Gunjan	Sambhal					
5	Sumit	Khurja					
6	Arpita	Lucknow					
7	Vaishali	Punjab					
8	Anisha	Shamili					
9	Ginni	Punjab					

Note: *Both the files will be saved in the current working directory of the R workspace.*

Reading the Excel File

Like the CSV file, we can read data from an excel file. R provides read.xlsx() function, which takes two arguments as input, i.e., file name and index of the sheet. This function returns the excel data in the form of a data frame in the R environment. There is the following syntax of read.xlsx() function:

```
read.xlsx(file_name,sheet_index)
```

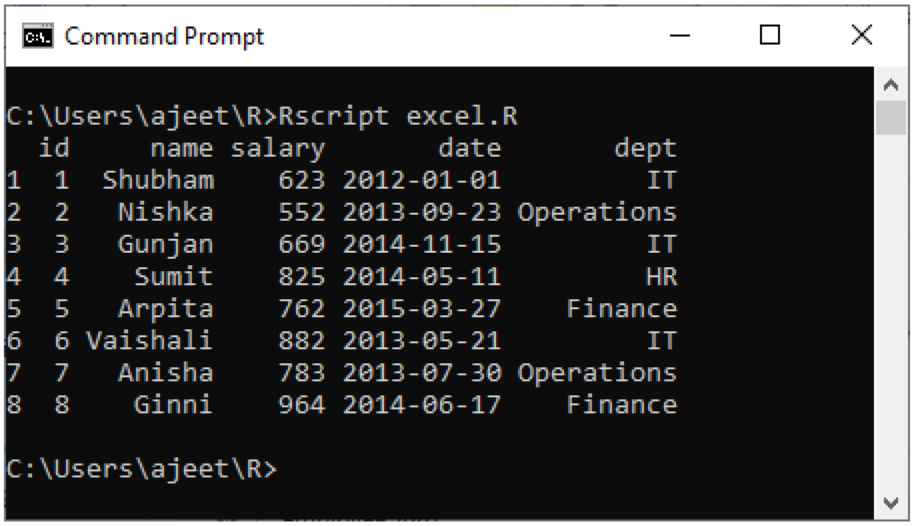
Let's see an example in which we read data from our employee.xlsx file.

Example

```
#Loading xlsx package
library("xlsx")
```

```
# Reading the first worksheet in the file employee.xlsx.
excel_data<- read.xlsx("employee.xlsx", sheetIndex = 1)
print(excel_data)
```

Output



```
C:\Users\ajeet\R>Rscript excel.R
  id     name salary       date       dept
1  1  Shubham    623 2012-01-01         IT
2  2   Nishka    552 2013-09-23 Operations
3  3   Gunjan    669 2014-11-15         IT
4  4    Sumit    825 2014-05-11         HR
5  5   Arpita    762 2015-03-27    Finance
6  6 Vaishali    882 2013-05-21         IT
7  7   Anisha    783 2013-07-30 Operations
8  8    Ginni    964 2014-06-17    Finance

C:\Users\ajeet\R>
```

Writing data into Excel File

In R, we can also write the data into our .xlsx file. R provides a write.xlsx() function to write data into the excel file. There is the following syntax of write.xlsx() function:

```
1. write.xlsx(data_frame,file_name,col.names,row.names,sheetnames,append)
```

Here,

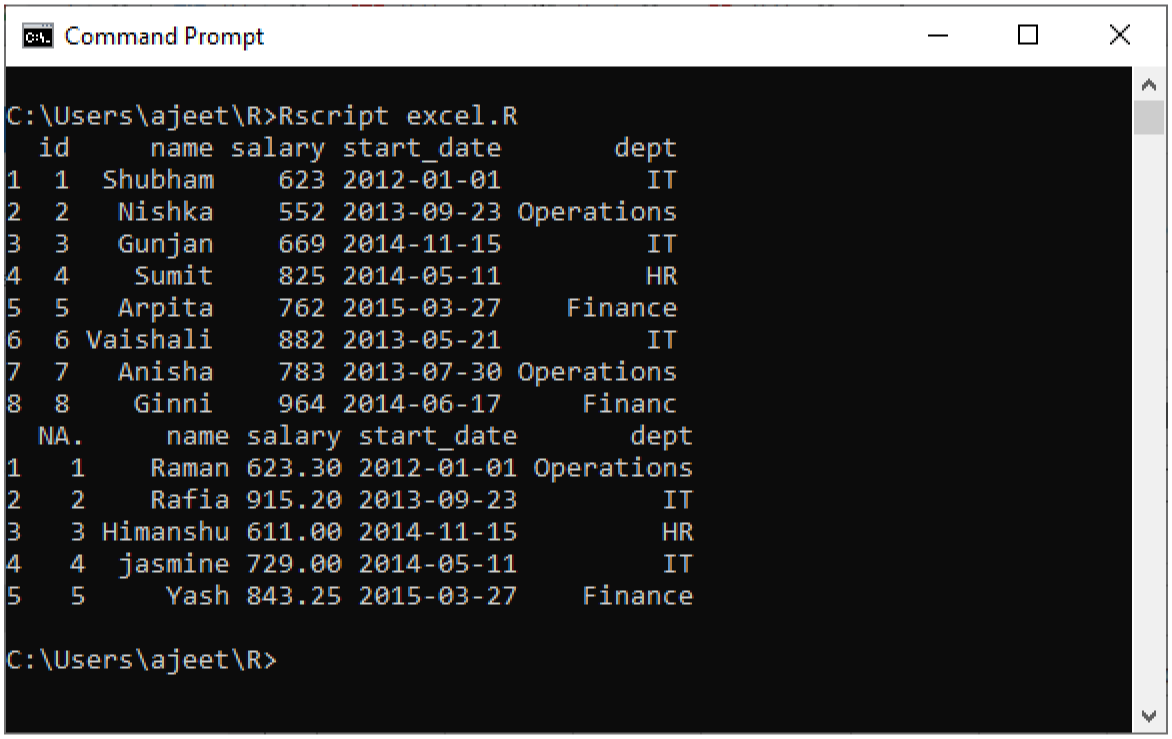
- The data_frame is our data, which we want to insert into our excel file.
- The file_names is the name of that file in which we want to insert our data.
- The col.names and row.names are the logical values that are specifying whether the column names/row names of the data frame are to be written to the file.
- The append is a logical value, which indicates our data should be appended or not into an existing file.

Let's see an example to understand how write.xlsx() function works with its parameters.

Example

```
#Loading xlsx package
library("xlsx")

#Creating data frame
emp.data<- data.frame(
name = c("Raman","Rafia","Himanshu","jasmine","Yash"),
salary = c(623.3,915.2,611.0,729.0,843.25),
start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11","2015-03-27")),
dept = c("Operations","IT","HR","IT","Finance"),
stringsAsFactors = FALSE    )


# Writing the first data set in employee.xlsxRscript
write.xlsx(emp.data, file = "employee.xlsx", col.names=TRUE, row.names=TRUE,sheetName="She
t2",append = TRUE)

# Reading the first worksheet in the file employee.xlsx.
excel_data<- read.xlsx("employee.xlsx", sheetIndex = 1)
print(excel_data)

# Reading the first worksheet in the file employee.xlsx.
excel_data<- read.xlsx("employee.xlsx", sheetIndex = 2)
print(excel_data)
```

Output



```
C:\Users\ajeet\R>Rscript excel.R
  id      name salary start_date       dept
1  1   Shubham    623 2012-01-01         IT
2  2    Nishka    552 2013-09-23 Operations
3  3    Gunjan    669 2014-11-15         IT
4  4     Sumit    825 2014-05-11         HR
5  5    Arpita    762 2015-03-27    Finance
6  6 Vaishali    882 2013-05-21         IT
7  7    Anisha    783 2013-07-30 Operations
8  8     Ginni    964 2014-06-17     Financ
  NA.      name salary start_date       dept
1   1     Raman 623.30 2012-01-01 Operations
2   2     Rafia 915.20 2013-09-23         IT
3   3 Himanshu 611.00 2014-11-15         HR
4   4   jasmine 729.00 2014-05-11         IT
5   5      Yash 843.25 2015-03-27    Finance

C:\Users\ajeet\R>
```