Foundation of R

Introduction

- R is a programming language developed by Ross Ihaka and Robert Gentleman in 1993.
- R possesses an extensive catalog of statistical and graphical methods.
- It includes machine learning algorithm, linear regression, time series, statistical inference to name a few.
- Most of the R libraries are written in R, but for heavy computational task, C, C++ and Fortran codes are preferred.
- All the libraries of R, almost 12k, are stored in CRAN. CRAN is a free and open source. You can download and use the numerous libraries to perform Machine Learning or time series analysis.

Data analysis with R is done in a series of steps:

- programming, transforming, discovering, modeling and communicate the results
- **Program**: R is a clear and accessible programming tool
- **Transform**: R is made up of a collection of libraries designed specifically for data science
- **Discover**: Investigate the data, refine your hypothesis and analyze them
- **Model**: R provides a wide array of tools to capture the right model for your data
- **Communicate**: Integrate codes, graphs, and outputs to a report with R Markdown or build Shiny apps to share with the world

What is R used for?

- Statistical inference
- Data analysis
- Machine learning algorithm

R Data Types

- R Programming works with numerous data types, including
  - Scalars
  - Vectors (numerical, character, logical)
  - Matrices
  - Data frames
  - Lists

Example 1:

```
# Declare variables of different types
# Numeric
x <- 28
class(x)
```

Output:

```
## [1] "numeric"
```

Example 2:

```
# String
y <- "R is Fantastic"
class(y)
```

Output:

```
## [1] "character"
```

Variables

- Variables store values and are an important component in programming, especially for a data scientist.
- A variable can store a number, an object, a statistical result, vector, dataset, a model prediction basically anything R outputs.
- To add a value to the variable, use <- or =.

  Here is the syntax:

Example 1:

```
# Print variable x
x <- 42
x
```

Output:

```
## [1] 42
```

Example 2:

```
y  <- 10
y
```

Output:

```
## [1] 10
```

Vectors

- A vector is a one-dimensional array.
- We can create a vector with all the basic data type we learnt before.
- The simplest way to build a vector in R, is to use the c command.

Example 1:

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

Output:

```
## [1]  1 10 49
```

Example 2:

```
# Character
vec_chr <- c("a", "b", "c")
vec_chr
```

Output:

```
## [1] "a" "b" "c"
```

Example 3:

```
# Boolean
vec_bool <-  c(TRUE, FALSE, TRUE)
vec_bool
```

Output:

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

We can do arithmetic calculations on vectors.

Example 4:

```
# Create the vectors
vect_1 <- c(1, 3, 5)
vect_2 <- c(2, 4, 6)
# Take the sum of A_vector and B_vector
sum_vect <- vect_1 + vect_2
# Print out total_vector
sum_vect
```

Output:

```
[1]  3  7 11
```

Example 5:

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

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

Output:

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

Example 6:

The shortest way to create a range of value is to use the: between two numbers. For instance, from the above example, we can write c(1:10) to create a vector of value from one to ten.

```
# Faster way to create adjacent values
c(1:10)
```

Output:

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

Arithmetic Operators

Arithmetic Operators

We will first see the basic arithmetic operations in R. The following operators stand for:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or **	Exponentiation

Example 1:

```
# An addition
3 + 4
```

Output:

```
## [1] 7
```

Logical Operators

- With logical operators, we want to return values inside the vector based on logical conditions. Following is a detailed list of logical operators available in R

- The logical statements in R are wrapped inside the []. We can add many conditional statements as we like but we need to include them in a parenthesis. We can follow this structure to create a conditional statement:
- variable_name[(conditional_statement)]

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

Example 1:

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

Output:

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

Example 2:

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

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

Output:

```
## [1]  6  7  8  9 10
```

Example 3:

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

Output:

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

What is a Matrix?

- A matrix is a 2-dimensional array that has m number of rows and n number of columns. In other words, matrix is a combination of two or more vectors with the same data type.
- **Note:** It is possible to create more than two dimensions arrays with R.
- We can create a matrix with the function matrix(). This function takes three arguments:

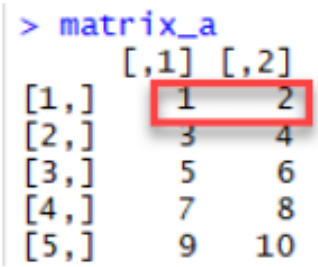
  Syntax : matrix(data, nrow, ncol, byrow = FALSE)

Arguments:

- **data**: The collection of elements that R will arrange into the rows and columns of the matrix \
- **nrow**: Number of rows
- **ncol**: Number of columns
- **byrow**: The rows are filled from the left to the right. We use `byrow = FALSE` (default values), if we want the matrix to be filled by the columns i.e. the values are filled top to bottom.

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

Output:



Print dimension of the matrix with dim()

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

Output:

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

Construct a matrix with 5 rows that contain the numbers 1 up to 10 and byrow = FALSE

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

Output:

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

Print dimension of the matrix with dim()

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

Output:

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

Add a Column to a Matrix with the cbind()

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

Example:

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

Output:

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

cbind()concatenate columns, rbind() appends rows. Let's add one row to our matrix_c matrix and verify the dimension is 5x3

```
matrix_c <-matrix(1:12, byrow = FALSE, ncol = 3)
# Create a vector of 3 columns
add_row <- c(1:3)
# Append to the matrix
matrix_c <- rbind(matrix_c, add_row)
# Check the dimension
dim(matrix_c)
```

Output:

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

Slice a Matrix

We can select elements one or many elements from a matrix by using the square brackets [ ]. This is where slicing comes into the picture.

For example:

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

What is Factor in R?

- **Categorical Variables**

R stores categorical variables into a factor. Let's check the code below to convert a character variable into a factor variable.

Syntax

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

Arguments:

- **x**: A vector of data. Need to be a string or integer, not decimal.
- **Levels**: A vector of possible values taken by x. This argument is optional. The default value is the unique list of items of the vector x.
- **Labels**: Add a label to the x data. For example, 1 can take the label `male` while 0, the label `female`.
- **ordered**: Determine if the levels should be ordered.

Let's create a factor data frame.

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

Output:

```
## [1] "character"
## [1] "factor"
```

It is important to transform a **string** into factor when we perform Machine Learning task.

A categorical variable can be divided into **nominal categorical variable** and **ordinal categorical variable**.

Nominal Categorical Variable

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

```
# Create a color vector
color_vector <- c('blue', 'red', 'green', 'white', 'black', 'yellow')
# Convert the vector to factor
factor_color <- factor(color_vector)
factor_color
```

Output:

```
## [1] blue   red    green  white  black  yellow
## Levels: black blue green red white yellow
```

Ordinal Categorical Variable

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

Example:

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

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

Output:

```
## [1] evening   morning   afternoon midday
midnight  evening
```

Example:

```
## Levels: morning < midday < afternoon < evening < midnight
# Append the line to above code
# Count the number of occurence of each level
summary(factor_day)
```

Output:

```
##   morning    midday afternoon   evening  midnight
##         1         1         1         2         1
```

R ordered the level from 'morning' to 'midnight' as specified in the levels parenthesis.

R Data Frame: Create, Append, Select, Subset

What is a Data Frame?

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

How to Create a Data Frame

We can create a data frame by passing the variable a,b,c,d into the data.frame() function. We can name the columns with name() and simply specify the name of the variables.

```
data.frame(df, stringsAsFactors = TRUE)
```

Arguments:

- **df**: It can be a matrix to convert as a data frame or a collection of variables to join
- **stringsAsFactors**: Convert string to factor by default

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

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

Output:

```
## 'data.frame':    4 obs. of  4 variables:
##  $ ID    : num  10 20 30 40
##  $ items: Factor w/ 4 levels "book","pen","pencil_case",..: 1 2 4 3
##  $ store: logi  TRUE FALSE TRUE FALSE
##  $ price: num  2.5 8 10 7
```

```
# Print the structure
str(df)
```

Output:

```
## 'data.frame':    4 obs. of  4 variables:
##  $ ID    : num  10 20 30 40
##  $ items: Factor w/ 4 levels "book","pen","pencil case",..: 1 2 4 3
```

Slice Data Frame

It is possible to SLICE values of a Data Frame. We select the rows and columns to return into bracket precede by the name of the data frame.

A data frame is composed of rows and columns, df[A, B]. A represents the rows and B the columns. We can slice either by specifying the rows and/or columns.

```
## Select Rows 1 to 2
df[1:2,]
```

Output:

```
##   ID items store price
## 1 10  book  TRUE   2.5
## 2 20   pen FALSE   8.0
```

```
## Select Rows 1 to 3 and columns 3 to 4
df[1:3, 3:4]
```

Output:

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

It is also possible to select the columns with their names. For instance, the code below extracts two columns: ID and store.

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

Output:

```
##   ID store
## 1 10  TRUE
## 2 20 FALSE
## 3 30  TRUE
## 4 40 FALSE
```

Append a Column to Data Frame

- You can also append a column to a Data Frame. You need to use the symbol $ to append a new variable.

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

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

Output:

```
##   ID       items store price quantity
## 1 10        book  TRUE   2.5       10
## 2 20         pen FALSE   8.0       35
## 3 30    textbook  TRUE  10.0       40
## 4 40 pencil_case FALSE   7.0        5
```

Select a Column of a Data Frame

- Sometimes, we need to store a column of a data frame for future use or perform operation on a column. We can use the $ sign to select the column from a data frame.

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

Output:

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

Subset a Data Frame

- In the previous slide, we selected an entire column without condition.
- It is possible to **subset** based on whether or not a certain condition was true.
- We use the subset() function.

```
subset(x, condition)
arguments:
- x: data frame used to perform the subset
- condition: define the conditional statement
```

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

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

Output:

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

What is a List?

- A **list** is a great tool to store many kinds of object in the order expected.
- We can include matrices, vectors data frames or lists.
- We can imagine a list as a bag in which we want to put many different items.
- When we need to use an item, we open the bag and use it. A list is similar; we can store a collection of objects and use them when we need them.

How to Create a List

We can use list() function to create a list.

```
list(element_1, ...)
arguments:
-element_1: store any type of R object
-...: pass as many objects as specifying. each object needs to be separated by a comma
```

In the example below, we create three different objects, a vector, a matrix and a data frame.

Step 1) Create a Vector

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

Step 2) Create a Matrices

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

Output:

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

Step 4) Create a List

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

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

Output:

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

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

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

Select Elements from List

After we built our list, we can access it quite easily. We need to use the [[index]] to select an element in a list. The value inside the double square bracket represents the position of the item in a list we want to extract. For instance, we pass 2 inside the parenthesis, R returns the second element listed.

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

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

Output:

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

Built-in Data Frame

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

```
PATH <-'https://raw.githubusercontent.com/guru99-edu/R-Programming/master/prison.csv'
df <- read.csv(PATH)[1:5]
head(df, 5)
```

Output:

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

We can check the structure of the data frame with str:

```
# Structure of the data
str(df)
```

Output:

```
## 'data.frame':    714 obs. of  5 variables:
##  $ X      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ state  : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ year   : int  80 81 82 83 84 85 86 87 88 89 ...
##  $ govelec: int  0 0 1 0 0 0 1 0 0 0 ...
##  $ black  : num  0.256 0.256 0.255 0.255 0.255 ...
```

All variables are stored in the **numerical** format.

What is a Function in R?

- A **function**, in a programming environment, is a set of instructions. A programmer builds a function to avoid **repeating the** same task, or reduce **complexity.**
- A function should be
  - ✓ written to carry out a specified a tasks
  - ✓ may or may not include arguments
  - ✓ contain a body
  - ✓ may or may not return one or more values.

A general approach to a function is to use the argument part as **inputs**, feed the **body** part and finally return an **output**. The Syntax of a function is the following:

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

Math functions

R has an array of mathematical functions.

Operator	Description
abs(x)	Takes the absolute value of x
log(x,base=y)	Takes the logarithm of x with base y; if base is not specified, returns the natural logarithm
exp(x)	Returns the exponential of x
sqrt(x)	Returns the square root of x
factorial(x)	Returns the factorial of x (x!)

```
# sequence of number from 44 to 55 both including incremented by 1
x_vector <- seq(45,55, by = 1)
#logarithm
log(x_vector)
```

Output:

```
##  [1] 3.806662 3.828641 3.850148 3.871201 3.891820 3.912023 3.931826
##  [8] 3.951244 3.970292 3.988984 4.007333
```

```
#exponential
exp(x_vector)
```

```
#squared root
sqrt(x_vector)
```

Output:

```
##  [1] 6.708204 6.782330 6.855655 6.928203 7.000000 7.071068 7.141428
##  [8] 7.211103 7.280110 7.348469 7.416198
```

Statistical functions

- R standard installation contains wide range of statistical functions. In this tutorial, we will briefly look at the most important function..

Basic statistic functions

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

```
speed <- dt$speed
speed
# Mean speed of cars dataset
mean(speed)
```

Output:

```
## [1] 15.4
```

```
# Median speed of cars dataset
median(speed)
```

Output:

```
## [1] 15
```

```
# Variance speed of cars dataset
var(speed)
```

Output:

```
## [1] 27.95918
```

```
# Standard deviation speed of cars dataset
sd(speed)
```

Output:

```
## [1] 5.287644
```

```
# Standardize vector speed of cars dataset
head(scale(speed), 5)
```

Output:

```
##           [,1]
## [1,] -2.155969
## [2,] -2.155969
## [3,] -1.588609
## [4,] -1.588609
## [5,] -1.399489
```

```
# Quantile speed of cars dataset
quantile(speed)
```

Output:

```
##   0%  25%  50%  75% 100%
##    4   12   15   19   25
```

```
# Summary speed of cars dataset
summary(speed)
```

Output:

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     4.0    12.0    15.0    15.4    19.0    25.0
```

Up to this point, we have learned a lot of R built-in functions.

Note: Be careful with the class of the argument, i.e. numeric, Boolean or string. For instance, if we need to pass a string value, we need to enclose the string in quotation mark: "ABC" .

Write function in R

In some occasion, we need to write our own function because we have to accomplish a particular task and no ready made function exists. A user-defined function involves a **name**, **arguments** and a **body**.

```
function.name <- function(arguments)
{
    computations on the arguments
    some other code
}
```

Note: A good practice is to name a user-defined function different from a built-in function. It avoids confusion.

One argument function

In the next snippet, we define a simple square function. The function accepts a value and returns the square of the value.

```
square_function<- function(n)
{
  # compute the square of integer `n`
  n^2
}
# calling the function and passing value 4
square_function(4)
```

Multi arguments function

We can write a function with more than one argument. Consider the function called "times". It is a straightforward function multiplying two variables.

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

Output:

```
## [1] 8
```

When should we write function?

Data scientist need to do many repetitive tasks. Most of the time, we copy and paste chunks of code repetitively. For example, normalization of a variable is highly recommended before we run a machine learning algorithm. The formula to normalize a variable is:

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

We already know how to use the min() and max() function in R. We use the tibble library to create the data frame. Tibble is so far the most convenient function to create a data set from scratch.

```
library(tibble)
# Create a data frame
data_frame <- tibble(
  c1 = rnorm(50, 5, 1.5),
  c2 = rnorm(50, 5, 1.5),
  c3 = rnorm(50, 5, 1.5),
)
```

```
normalize <- function(x){
  # step 1: create the nominator
  nominator <- x-min(x)
  # step 2: create the denominator
  denominator <- max(x)-min(x)
  # step 3: divide nominator by denominator
  normalize <- nominator/denominator
  # return the value
  return(normalize)
}
```

Let's test our function with the variable c1:

```
normalize(data_frame$c1)
```

It works perfectly. We created our first function.

Functions are more comprehensive way to perform a repetitive task. We can use the normalize formula over different columns, like below:

```
data_frame$c1_norm_function <- normalize (data_frame$c1)
data_frame$c2_norm_function <- normalize        (data_frame$c2)
data_frame$c3_norm_function <- normalize        (data_frame$c3)
```

Functions with condition

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

Let's try our function on the airquality dataset. we should have one train set with 122 rows and a test set with 31 rows.

```
train <- split_data(airquality, train = TRUE)
dim(train)
```

The if else statement

An if-else statement is a great tool for the developer trying to return an output based on a condition. In R, the syntax is:

```
if (condition) {
    Expr1
} else {
    Expr2
}
```

```
# Create vector quantity
quantity <-  25
# Set the is-else statement
if (quantity > 20) {
    print('You sold a lot!')
} else {
    print('Not enough for today')
}
```

Output:

```
## [1] "You sold a lot!"
```

The else if statement

We can further customize the control level with the else if statement. With elif, you can add as many conditions as we want. The syntax is:

```
if (condition1) {
    expr1
    } else if (condition2) {
    expr2
    } else if  (condition3) {
    expr3
    } else {
    expr4
}
```

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

Output:

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

```
# Create vector quantiy
quantity <-  10
# Create multiple condition statement
if (quantity <20) {
      print('Not enough for today')
} else if (quantity > 20  &quantity <= 30) {
     print('Average day')
} else {
      print('What a great day!')
}
```

Output:

```
## [1] "Not enough for today"
```

For Loop in R with Examples for List and Matrix

- A for loop is very valuable when we need to iterate over a list of elements or a range of numbers. Loop can be used to iterate over a list, data frame, vector, matrix or any other object. The braces and square bracket are compulsory.

For Loop Syntax and Examples

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

Example 1: We iterate over all the elements of a vector and print the current value.

```
# Create fruit vector
fruit <- c('Apple', 'Orange', 'Passion fruit', 'Banana')
# Create the for statement
for ( i in fruit){
 print(i)
}
```

Output:

```
## [1] "Apple"
## [1] "Orange"
## [1] "Passion fruit"
## [1] "Banana"
```

Example 2: creates a non-linear function by using the polynomial of x between 1 and 4 and we store it in a list

```
# Create an empty list
list <- c()
# Create a for statement to populate the list
for (i in seq(1, 4, by=1)) {
  list[[i]] <- i*i
}
print(list)
```

Output:

```
## [1]  1  4 9 16
```

For Loop over a list

Looping over a list is just as easy and convenient as looping over a vector. Let's see an example

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

Output:

```
## [1] "Apple" "Orange" "Passion fruit" "Banana"
## [1] 10 12 15
## [1] FALSE
```

For Loop over a matrix

A matrix has 2-dimension, rows and columns. To iterate over a matrix, we have to define two for loop, namely one for the rows and another for the column.

```
# Create a matrix
mat <- matrix(data = seq(10, 20, by=1), nrow = 6, ncol =2)
# Create the loop with r and c to iterate over the matrix
for (r in 1:nrow(mat))
    for (c in 1:ncol(mat))
         print(paste("Row", r, "and column",c, "have values of", mat[r,c]))
```

Output:

```
## [1] "Row 1 and column 1 have values of 10"
## [1] "Row 1 and column 2 have values of 16"
## [1] "Row 2 and column 1 have values of 11"
## [1] "Row 2 and column 2 have values of 17"
## [1] "Row 3 and column 1 have values of 12"
## [1] "Row 3 and column 2 have values of 18"
## [1] "Row 4 and column 1 have values of 13"
## [1] "Row 4 and column 2 have values of 19"
## [1] "Row 5 and column 1 have values of 14"
## [1] "Row 5 and column 2 have values of 20"
## [1] "Row 6 and column 1 have values of 15"
## [1] "Row 6 and column 2 have values of 10"
```

While Loop in R with Example

A loop is a statement that keeps running until a condition is satisfied. The syntax for a while loop is the following:

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

Example 1:

Let's go through a very simple example to understand the concept of while loop. You will create a loop and after each run add 1 to the stored variable. You need to close the loop, therefore we explicitely tells R to stop looping when the variable reached 10.

Note: If you want to see current loop value, you need to wrap the variable inside the function print().

```
#Create a variable with value 1
begin <- 1

#Create the loop
while (begin <= 10){

#See which we are
cat('This is loop number',begin)

#add 1 to the variable begin after each loop
begin <- begin+1
print(begin)
}
```

Input as CSV File

- The csv file is a text file in which the values in the columns are separated by a comma. Let's consider the following data present in the file named **input.csv.**

Reading a CSV File

Following is a simple example of **read.csv()** function to read a CSV file available in your current working directory –

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

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

```
      id,    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     NA     Gary     843.25     2015-03-27      Finance
6      6     Nina     578.00     2013-05-21      IT
7      7     Simon    632.80     2013-07-30      Operations
8      8     Guru     722.50     2014-06-17      Finance
```

Analyzing the CSV File

By default the **read.csv()** function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

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

print(is.data.frame(data))
print(ncol(data))
print(nrow(data))
```

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

```
[1] TRUE
[1] 5
[1] 8
```

Once we read data in a data frame, we can apply all the functions applicable to data frames as explained in subsequent section.

Get the details of the person with max salary

We can fetch rows meeting specific filter criteria similar to a SQL where clause.

```
# Create a data frame.
data <- read.csv("input.csv")

# Get the max salary from data frame.
sal <- max(data$salary)

# Get the person detail having max salary.
retval <- subset(data, salary == max(salary))
print(retval)
```

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

```
      id    name  salary  start_date    dept
5     NA    Gary  843.25  2015-03-27    Finance
```

Get the persons in IT department whose salary is greater than 600

```
# Create a data frame.
data <- read.csv("input.csv")

info <- subset(data, salary > 600 & dept == "IT")
print(info)
```

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

```
      id   name      salary   start_date   dept
1      1   Rick      623.3    2012-01-01   IT
3      3   Michelle  611.0    2014-11-15   IT
```

Get the people who joined on or after 2014

```
# Create a data frame.
data <- read.csv("input.csv")

retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
print(retval)
```

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

```
      id   name      salary   start_date    dept
3      3   Michelle 611.00   2014-11-15    IT
4      4   Ryan     729.00   2014-05-11    HR
5     NA   Gary     843.25   2015-03-27    Finance
8      8   Guru     722.50   2014-06-17    Finance
```

Writing into a CSV File

R can create csv file form existing data frame. The **write.csv()** function is used to create the csv file. This file gets created in the working directory.

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.
write.csv(retval,"output.csv")
newdata <- read.csv("output.csv")
print(newdata)
```

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

```
  X      id   name      salary   start_date    dept
1 3      3    Michelle  611.00   2014-11-15    IT
2 4      4    Ryan      729.00   2014-05-11    HR
3 5     NA    Gary      843.25   2015-03-27    Finance
4 8      8    Guru      722.50   2014-06-17    Finance
```

Here the column X comes from the data set newper. This can be dropped using additional parameters while writing the file.

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.
write.csv(retval,"output.csv", row.names = FALSE)
newdata <- read.csv("output.csv")
print(newdata)
```

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

```
      id    name      salary   start_date    dept
1      3    Michelle  611.00   2014-11-15    IT
2      4    Ryan      729.00   2014-05-11    HR
3     NA    Gary      843.25   2015-03-27    Finance
4      8    Guru      722.50   2014-06-17    Finance
```

R - Pie Charts

Syntax

The basic syntax for creating a pie-chart using the R is −

```
pie(x, labels, radius, main, col, clockwise)
```

Following is the description of the parameters used −

- **x** is a vector containing the numeric values used in the pie chart.
- **labels** is used to give description to the slices.
- **radius** indicates the radius of the circle of the pie chart.(value between −1 and +1).
- **main** indicates the title of the chart.
- **col** indicates the color palette.
- **clockwise** is a logical value indicating if the slices are drawn clockwise or anti clockwise.

Slice Percentages and Chart Legend

We can add slice percentage and a chart legend by creating additional chart variables.

Live Demo

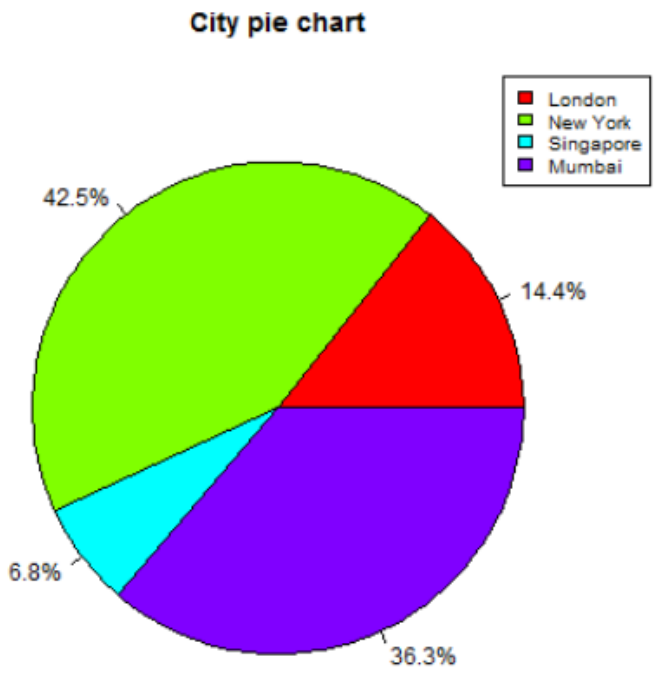
```
# Create data for the graph.
x <-  c(21, 62, 10,53)
labels <-  c("London","New York","Singapore","Mumbai")

piepercent<- round(100*x/sum(x), 1)

# Give the chart file a name.
png(file = "city_percentage_legends.jpg")

# Plot the chart.
pie(x, labels = piepercent, main = "City pie chart",col = rainbow(length(x)))
legend("topright", c("London","New York","Singapore","Mumbai"), cex = 0.8,
   fill = rainbow(length(x)))

# Save the file.
dev.off()
```



Syntax

The basic syntax to create a bar-chart in R is −

```
barplot(H,xlab,ylab,main, names.arg,col)
```

Following is the description of the parameters used −

- **H** is a vector or matrix containing numeric values used in bar chart.
- **xlab** is the label for x axis.
- **ylab** is the label for y axis.
- **main** is the title of the bar chart.
- **names.arg** is a vector of names appearing under each bar.
- **col** is used to give colors to the bars in the graph.

Bar Chart Labels, Title and Colors

The features of the bar chart can be expanded by adding more parameters. The **main** parameter is used to add **title**. The **col** parameter is used to add colors to the bars. The **args.name** is a vector having same number of values as the input vector to describe the meaning of each bar.

Example

The below script will create and save the bar chart in the current R working directory.

Live Demo

```
# Create the data for the chart
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul")

# Give the chart file a name
png(file = "barchart_months_revenue.png")

# Plot the bar chart
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",
main="Revenue chart",border="red")

# Save the file
```