

# UNIT - 4 & 5

## R - Overview

R is a programming language and software environment for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.

The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions. R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

R is free software distributed under a GNU-style copy left, and an official part of the GNU project called **GNU S**.

### Evolution of R

R was initially written by **Ross Ihaka** and **Robert Gentleman** at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.

- A large group of individuals has contributed to R by sending code and bug reports.
- Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.

### Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R –

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

As a conclusion, R is world's most widely used statistics programming language. It's the # 1 choice of data scientists and supported by a vibrant and talented community .

## R - Basic Syntax

As a convention, we will start learning R programming by writing a "Hello, World!" program. Depending on the needs, you can program either at R command prompt or you can use an R script file to write your program. Let's check both one by one.

### R Command Prompt

Once you have R environment setup, then it's easy to start your R command prompt by just typing the following command at your command prompt –

```
$ R
```

This will launch R interpreter and you will get a prompt > where you can start typing your program as follows –

```
> myString <- "Hello, World!"
> print ( myString)
[1] "Hello, World!"
```

Here first statement defines a string variable myString, where we assign a string "Hello, World!" and then next statement print() is being used to print the value stored in variable myString.

## R Script File

Usually, you will do your programming by writing your programs in script files and then you execute those scripts at your command prompt with the help of R interpreter called **Rscript**. So let's start with writing following code in a text file called test.R as under –

[Live Demo](#)

```
# My first program in R Programming
myString <- "Hello, World!"

print ( myString)
```

Save the above code in a file test.R and execute it at Linux command prompt as given below. Even if you are using Windows or other system, syntax will remain same.

```
$ Rscript test.R
```

When we run the above program, it produces the following result.

```
[1] "Hello, World!"
```

## Comments

Comments are like helping text in your R program and they are ignored by the interpreter while executing your actual program. Single comment is written using # in the beginning of the statement as follows –

```
# My first program in R Programming
```

R does not support multi-line comments but you can perform a trick which is something as follows –

[Live Demo](#)

```
if(FALSE) {
    "This is a demo for multi-line comments and it should be put inside either a
    single OR double quote"
}

myString <- "Hello, World!"
print ( myString)
[1] "Hello, World!"
```

Though above comments will be executed by R interpreter, they will not interfere with your actual program. You should put such comments inside, either single or double quote.

## R - Data Types

Generally, while doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that, when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

In contrast to other programming languages like C and java in R, the variables are not declared as some data type. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are –

- Vectors
- Lists
- Matrices
- Arrays
- Factors
- Data Frames

The simplest of these objects is the **vector object** and there are six data types of these atomic vectors, also termed as six classes of vectors. The other R-Objects are built upon the atomic vectors.

Data Type	Example	Verify
Logical	TRUE, FALSE	<a href="#">Live Demo</a> <pre>v &lt;- TRUE print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "logical"</pre>
Numeric	12.3, 5, 999	<a href="#">Live Demo</a> <pre>v &lt;- 23.5 print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "numeric"</pre>
Integer	2L, 34L, 0L	<a href="#">Live Demo</a> <pre>v &lt;- 2L print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "integer"</pre>
Complex	3 + 2i	<a href="#">Live Demo</a> <pre>v &lt;- 2+5i</pre>

		<pre>print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "complex"</pre>
Character	'a', '"good"', 'TRUE', '23.4'	<div>Live Demo</div> <pre>v &lt;- "TRUE" print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "character"</pre>
Raw	"Hello" is stored as 48 65 6c 6c 6f	<div>Live Demo</div> <pre>v &lt;- charToRaw("Hello") print(class(v))</pre> <p>it produces the following result –</p> <pre>[1] "raw"</pre>

In R programming, the very basic data types are the R-objects called **vectors** which hold elements of different classes as shown above. Please note in R the number of classes is not confined to only the above six types. For example, we can use many atomic vectors and create an array whose class will become array.

## Vectors

When you want to create vector with more than one element, you should use **c()** function which means to combine the elements into a vector.

Live Demo

```
# Create a vector.
apple <- c('red', 'green', "yellow")
print(apple)

# Get the class of the vector.
print(class(apple))
```

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

```
[1] "red"      "green"    "yellow"
[1] "character"
```

## Lists

A list is an R-object which can contain many different types of elements inside it like vectors, functions and even another list inside it.

Live Demo

```
# Create a list.
list1 <- list(c(2, 5, 3), 21.3, sin)
```

```
# Print the list.  
print(list1)
```

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

```
[[1]]  
[1] 2 5 3
```

```
[[2]]  
[1] 21.3
```

```
[[3]]  
function (x)  .Primitive("sin")
```

## Matrices

A matrix is a two-dimensional rectangular data set. It can be created using a vector input to the matrix function.

[Live Demo](#)

```
# Create a matrix.  
M = matrix( c('a','a','b','c','b','a'), nrow = 2, ncol = 3, byrow = TRUE)  
print(M)
```

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

```
      [,1] [,2] [,3]  
[1,] "a"  "a"  "b"  
[2,] "c"  "b"  "a"
```

## Arrays

While matrices are confined to two dimensions, arrays can be of any number of dimensions. The array function takes a dim attribute which creates the required number of dimension. In the below example we create an array with two elements which are 3x3 matrices each.

[Live Demo](#)

```
# Create an array.  
a <- array(c('green','yellow'),dim = c(3,3,2))  
print(a)
```

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

```
, , 1
```

```
      [,1]      [,2]      [,3]  
[1,] "green"  "yellow" "green"  
[2,] "yellow" "green"  "yellow"  
[3,] "green"  "yellow" "green"
```

```
, , 2
```

```
      [,1]      [,2]      [,3]  
[1,] "yellow" "green"  "yellow"  
[2,] "green"  "yellow" "green"  
[3,] "yellow" "green"  "yellow"
```

## Factors

Factors are the R-objects which are created using a vector. It stores the vector along with the distinct values of the elements in the vector as labels. The labels are always character irrespective of whether it is numeric or character or Boolean etc. in the input vector. They are useful in statistical modeling.

Factors are created using the **factor()** function. The **nlevels** function gives the count of levels.

[Live Demo](#)

```
# Create a vector.
apple_colors <- c('green','green','yellow','red','red','red','green')

# Create a factor object.
factor_apple <- factor(apple_colors)

# Print the factor.
print(factor_apple)
print(nlevels(factor_apple))
```

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

```
[1] green green yellow red red red green
Levels: green red yellow
[1] 3
```

## Data Frames

Data frames are tabular data objects. Unlike a matrix in data frame each column can contain different modes of data. The first column can be numeric while the second column can be character and third column can be logical. It is a list of vectors of equal length.

Data Frames are created using the **data.frame()** function.

[Live Demo](#)

```
# Create the data frame.
BMI <- data.frame(
  gender = c("Male", "Male", "Female"),
  height = c(152, 171.5, 165),
  weight = c(81, 93, 78),
  Age = c(42, 38, 26)
)
print(BMI)
```

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

```
  gender height weight Age
1  Male   152.0     81  42
2  Male   171.5     93  38
3 Female   165.0     78  26
```

## R - Variables

A variable provides us with named storage that our programs can manipulate. A variable in R can store an atomic vector, group of atomic vectors or a combination of many R objects. A valid variable name consists of letters, numbers and the dot or underline characters. The variable name starts with a letter or the dot not followed by a number.

Variable Name	Validity	Reason
var_name2.	valid	Has letters, numbers, dot and underscore
var_name%	Invalid	Has the character '%'. Only dot(.) and underscore allowed.
2var_name	invalid	Starts with a number
.var_name, var.name	valid	Can start with a dot(.) but the dot(.)should not be followed by a number.
.2var_name	invalid	The starting dot is followed by a number making it invalid.
_var_name	invalid	Starts with _ which is not valid

## Variable Assignment

The variables can be assigned values using leftward, rightward and equal to operator. The values of the variables can be printed using **print()** or **cat()** function. The **cat()** function combines multiple items into a continuous print output.

[Live Demo](#)

```
# Assignment using equal operator.
var.1 = c(0,1,2,3)

# Assignment using leftward operator.
var.2 <- c("learn","R")

# Assignment using rightward operator.
c(TRUE,1) -> var.3

print(var.1)
cat ("var.1 is ", var.1 ,"\n")
cat ("var.2 is ", var.2 ,"\n")
cat ("var.3 is ", var.3 ,"\n")
```

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

```
[1] 0 1 2 3
var.1 is  0 1 2 3
var.2 is  learn R
var.3 is  1 1
```

**Note** – The vector `c(TRUE,1)` has a mix of logical and numeric class. So logical class is coerced to numeric class making TRUE as 1.

## Data Type of a Variable

In R, a variable itself is not declared of any data type, rather it gets the data type of the R - object assigned to it. So R is called a dynamically typed language, which means that we can change a variable's data type of the same variable again and again when using it in a program.

[Live Demo](#)

```
var_x <- "Hello"
cat("The class of var_x is ", class(var_x), "\n")

var_x <- 34.5
cat("  Now the class of var_x is ", class(var_x), "\n")

var_x <- 27L
cat("    Next the class of var_x becomes ", class(var_x), "\n")
```

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

```
The class of var_x is  character
  Now the class of var_x is  numeric
    Next the class of var_x becomes  integer
```

## Finding Variables

To know all the variables currently available in the workspace we use the **ls()** function. Also the **ls()** function can use patterns to match the variable names.

[Live Demo](#)

```
print(ls())
```

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

```
[1] "my var"      "my_new_var" "my_var"      "var.1"
[5] "var.2"      "var.3"      "var.name"    "var_name2."
[9] "var_x"      "varname"
```

**Note** – It is a sample output depending on what variables are declared in your environment.

The **ls()** function can use patterns to match the variable names.

[Live Demo](#)

```
# List the variables starting with the pattern "var".
print(ls(pattern = "var"))
```

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

```
[1] "my var"      "my_new_var" "my_var"      "var.1"
[5] "var.2"      "var.3"      "var.name"    "var_name2."
[9] "var_x"      "varname"
```

The variables starting with **dot(.)** are hidden, they can be listed using "all.names = TRUE" argument to **ls()** function.

[Live Demo](#)

```
print(ls(all.name = TRUE))
```



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

```
[1] ".cars"          ".Random.seed"  ".var_name"     ".varname"      ".varname2"
[6] "my var"         "my_new_var"    "my_var"        "var.1"         "var.2"
[11] "var.3"          "var.name"      "var_name2."    "var_x"
```

## Deleting Variables

Variables can be deleted by using the **rm()** function. Below we delete the variable var.3. On printing the value of the variable error is thrown.

[Live Demo](#)

```
rm(var.3)
print(var.3)
```

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

```
[1] "var.3"
Error in print(var.3) : object 'var.3' not found
```

All the variables can be deleted by using the **rm()** and **ls()** function together.

[Live Demo](#)

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

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

```
character(0)
```

## R - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

### Types of Operators

We have the following types of operators in R programming –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

### Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

Operator	Description	Example
+	Adds two vectors	<a href="#">Live Demo</a>

		<pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v+t)</pre> <p>it produces the following result –</p> <pre>[1] 10.0 8.5 10.0</pre>
-	Subtracts second vector from the first	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v-t)</pre> <p>it produces the following result –</p> <pre>[1] -6.0 2.5 2.0</pre>
*	Multiplies both vectors	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v*t)</pre> <p>it produces the following result –</p> <pre>[1] 16.0 16.5 24.0</pre>
/	Divide the first vector with the second	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v/t)</pre> <p>When we execute the above code, it produces the following result –</p> <pre>[1] 0.250000 1.833333 1.500000</pre>
%%	Give the remainder of the first vector with the second	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v%%t)</pre> <p>it produces the following result –</p> <pre>[1] 2.0 2.5 2.0</pre>
%%/%	The result of division of first vector with second (quotient)	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c( 2,5.5,6) t &lt;- c(8, 3, 4) print(v%%/%t)</pre> <p>it produces the following result –</p> <pre>[1] 0 1 1</pre>

^	The first vector raised to the exponent of second vector	<a href="#">Live Demo</a> <pre>v &lt;- c( 2, 5.5, 6) t &lt;- c(8, 3, 4) print(v^t)</pre> <p>it produces the following result –</p> <pre>[1] 256.000 166.375 1296.000</pre>
---	--	--

## Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	<a href="#">Live Demo</a> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v&gt;t)</pre> <p>it produces the following result –</p> <pre>[1] FALSE TRUE FALSE FALSE</pre>
<	Checks if each element of the first vector is less than the corresponding element of the second vector.	<a href="#">Live Demo</a> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v &lt; t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE FALSE TRUE FALSE</pre>
==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	<a href="#">Live Demo</a> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v == t)</pre> <p>it produces the following result –</p> <pre>[1] FALSE FALSE FALSE TRUE</pre>
<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	<a href="#">Live Demo</a> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v&lt;=t)</pre>

		<p>it produces the following result –</p> <pre>[1] TRUE FALSE TRUE TRUE</pre>
>=	<p>Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v &gt;= t)</pre> <p>it produces the following result –</p> <pre>[1] FALSE TRUE FALSE TRUE</pre>
!=	<p>Checks if each element of the first vector is unequal to the corresponding element of the second vector.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(2, 5.5, 6, 9) t &lt;- c(8, 2.5, 14, 9) print(v != t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE TRUE TRUE FALSE</pre>

## Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example
&	<p>It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(3, 1, TRUE, 2+3i) t &lt;- c(4, 1, FALSE, 2+3i) print(v &amp; t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE TRUE FALSE TRUE</pre>
	<p>It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.</p>	<p><a href="#">Live Demo</a></p> <pre>v &lt;- c(3, 0, TRUE, 2+2i) t &lt;- c(4, 0, FALSE, 2+3i) print(v   t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE FALSE TRUE TRUE</pre>

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	<a href="#">Live Demo</a> <pre>v &lt;- c(3,0,TRUE,2+2i) print(!v)</pre> <p>it produces the following result –</p> <pre>[1] FALSE TRUE FALSE FALSE</pre>
---	---	---

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example
&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	<a href="#">Live Demo</a> <pre>v &lt;- c(3,0,TRUE,2+2i) t &lt;- c(1,3,TRUE,2+3i) print(v&amp;&amp; t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE</pre>
	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	<a href="#">Live Demo</a> <pre>v &lt;- c(0,0,TRUE,2+2i) t &lt;- c(0,3,TRUE,2+3i) print(v   t)</pre> <p>it produces the following result –</p> <pre>[1] FALSE</pre>

## Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example
<- or = or <<-	Called Left Assignment	<a href="#">Live Demo</a> <pre>v1 &lt;- c(3,1,TRUE,2+3i) v2 &lt;&lt;- c(3,1,TRUE,2+3i) v3 = c(3,1,TRUE,2+3i) print(v1) print(v2) print(v3)</pre> <p>it produces the following result –</p> <pre>[1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i</pre>

		[1] 3+0i 1+0i 1+0i 2+3i
-> or ->>	Called Right Assignment	<div>Live Demo</div> <pre>c(3,1,TRUE,2+3i) -&gt; v1 c(3,1,TRUE,2+3i) -&gt;&gt; v2 print(v1) print(v2)</pre> <p>it produces the following result –</p> <pre>[1] 3+0i 1+0i 1+0i 2+3i [1] 3+0i 1+0i 1+0i 2+3i</pre>

## Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example
:	Colon operator. It creates the series of numbers in sequence for a vector.	<div>Live Demo</div> <pre>v &lt;- 2:8 print(v)</pre> <p>it produces the following result –</p> <pre>[1] 2 3 4 5 6 7 8</pre>
%in%	This operator is used to identify if an element belongs to a vector.	<div>Live Demo</div> <pre>v1 &lt;- 8 v2 &lt;- 12 t &lt;- 1:10 print(v1 %in% t) print(v2 %in% t)</pre> <p>it produces the following result –</p> <pre>[1] TRUE [1] FALSE</pre>
%*%	This operator is used to multiply a matrix with its transpose.	<div>Live Demo</div> <pre>M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE) t = M %*% t(M) print(t)</pre> <p>it produces the following result –</p> <pre>[,1] [,2]</pre>

		[1, ]	65	82
		[2, ]	82	117

## R - Decision making

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Following is the general form of a typical decision making structure found in most of the programming languages –

R provides the following types of decision making statements. Click the following links to check their detail.

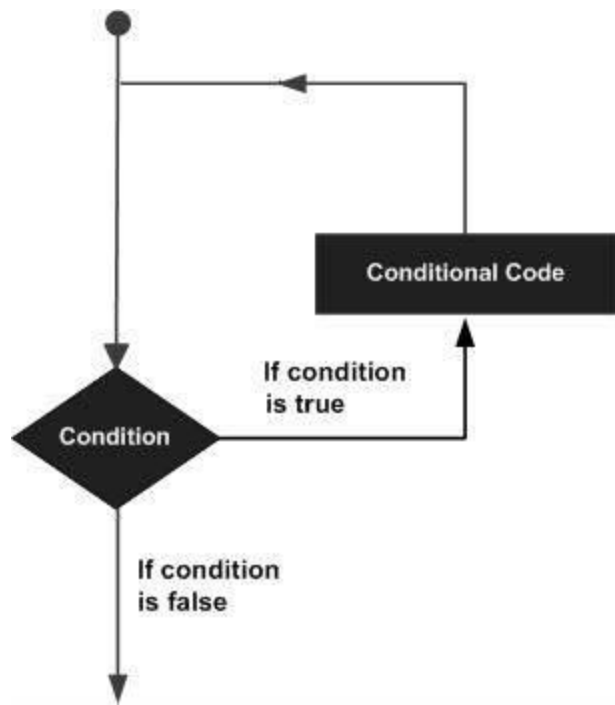
Sr.No.	Statement & Description
1	<a href="#">if statement</a>  An <b>if</b> statement consists of a Boolean expression followed by one or more statements.
2	<a href="#">if...else statement</a>  An <b>if</b> statement can be followed by an optional <b>else</b> statement, which executes when the Boolean expression is false.
3	<a href="#">switch statement</a>  A <b>switch</b> statement allows a variable to be tested for equality against a list of values.

## R - Loops

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages –



R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

Sr.No.	Loop Type & Description
1	<a href="#">repeat loop</a>  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2	<a href="#">while loop</a>  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	<a href="#">for loop</a>  Like a while statement, except that it tests the condition at the end of the loop body.

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	<a href="#">break statement</a>



	Terminates the <b>loop</b> statement and transfers execution to the statement immediately following the loop.
2	<u>Next statement</u>  The <b>next</b> statement simulates the behavior of R switch.

## R - Functions

A function is a set of statements organized together to perform a specific task. R has a large number of in-built functions and the user can create their own functions.

In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions.

The function in turn performs its task and returns control to the interpreter as well as any result which may be stored in other objects.

### Function Definition

An R function is created by using the keyword **function**. The basic syntax of an R function definition is as follows –

```
function_name <- function(arg_1, arg_2, ...) {
  Function body
}
```

### Function Components

The different parts of a function are –

- **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body** – The function body contains a collection of statements that defines what the function does.
- **Return Value** – The return value of a function is the last expression in the function body to be evaluated.

R has many **in-built** functions which can be directly called in the program without defining them first. We can also create and use our own functions referred as **user defined** functions.

### Built-in Function

Simple examples of in-built functions are **seq()**, **mean()**, **max()**, **sum(x)** and **paste(...)** etc. They are directly called by user written programs. You can refer [most widely used R functions](#).

[Live Demo](#)

```
# Create a sequence of numbers from 32 to 44.
print(seq(32,44))

# Find mean of numbers from 25 to 82.
```

```
print(mean(25:82))

# Find sum of numbers frm 41 to 68.
print(sum(41:68))
```

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

```
[1] 32 33 34 35 36 37 38 39 40 41 42 43 44
[1] 53.5
[1] 1526
```

## User-defined Function

We can create user-defined functions in R. They are specific to what a user wants and once created they can be used like the built-in functions. Below is an example of how a function is created and used.

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}
```

## Calling a Function

[Live Demo](#)

```
# Create a function to print squares of numbers in sequence.
new.function <- function(a) {
  for(i in 1:a) {
    b <- i^2
    print(b)
  }
}

# Call the function new.function supplying 6 as an argument.
new.function(6)
```

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

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

## Calling a Function without an Argument

[Live Demo](#)

```
# Create a function without an argument.
new.function <- function() {
  for(i in 1:5) {
    print(i^2)
  }
}
```

```
# Call the function without supplying an argument.  
new.function()
```

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

```
[1] 1  
[1] 4  
[1] 9  
[1] 16  
[1] 25
```

## Calling a Function with Argument Values (by position and by name)

The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

[Live Demo](#)

```
# Create a function with arguments.  
new.function <- function(a,b,c) {  
  result <- a * b + c  
  print(result)  
}  
  
# Call the function by position of arguments.  
new.function(5,3,11)  
  
# Call the function by names of the arguments.  
new.function(a = 11, b = 5, c = 3)
```

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

```
[1] 26  
[1] 58
```

## Calling a Function with Default Argument

We can define the value of the arguments in the function definition and call the function without supplying any argument to get the default result. But we can also call such functions by supplying new values of the argument and get non default result.

[Live Demo](#)

```
# Create a function with arguments.  
new.function <- function(a = 3, b = 6) {  
  result <- a * b  
  print(result)  
}  
  
# Call the function without giving any argument.  
new.function()  
  
# Call the function with giving new values of the argument.  
new.function(9,5)
```

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

```
[1] 18
```

## Lazy Evaluation of Function

Arguments to functions are evaluated lazily, which means so they are evaluated only when needed by the function body.

[Live Demo](#)

```
# Create a function with arguments.
new.function <- function(a, b) {
  print(a^2)
  print(a)
  print(b)
}

# Evaluate the function without supplying one of the arguments.
new.function(6)
```

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

```
[1] 36
[1] 6
Error in print(b) : argument "b" is missing, with no default
```

## R - Strings

Any value written within a pair of single quote or double quotes in R is treated as a string. Internally R stores every string within double quotes, even when you create them with single quote.

### Rules Applied in String Construction

- The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
- Double quotes can be inserted into a string starting and ending with single quote.
- Single quote can be inserted into a string starting and ending with double quotes.
- Double quotes can not be inserted into a string starting and ending with double quotes.
- Single quote can not be inserted into a string starting and ending with single quote.

### Examples of Valid Strings

Following examples clarify the rules about creating a string in R.

[Live Demo](#)

```
a <- 'Start and end with single quote'
print(a)

b <- "Start and end with double quotes"
print(b)

c <- "single quote ' in between double quotes"
print(c)

d <- 'Double quotes " in between single quote'
```

```
print(d)
```

When the above code is run we get the following output –

```
[1] "Start and end with single quote"
[1] "Start and end with double quotes"
[1] "single quote ' in between double quote"
[1] "Double quote \" in between single quote"
```

## Examples of Invalid Strings

[Live Demo](#)

```
e <- 'Mixed quotes'
print(e)

f <- 'Single quote ' inside single quote'
print(f)

g <- "Double quotes " inside double quotes"
print(g)
```

When we run the script it fails giving below results.

```
Error: unexpected symbol in:
"print(e)
f <- 'Single"
Execution halted
```

## String Manipulation

### Concatenating Strings - paste() function

Many strings in R are combined using the **paste()** function. It can take any number of arguments to be combined together.

### Syntax

The basic syntax for paste function is –

```
paste(..., sep = " ", collapse = NULL)
```

Following is the description of the parameters used –

- ... represents any number of arguments to be combined.
- **sep** represents any separator between the arguments. It is optional.
- **collapse** is used to eliminate the space in between two strings. But not the space within two words of one string.

### Example

[Live Demo](#)

```
a <- "Hello"
b <- 'How'
c <- "are you? "
```

```
print(paste(a,b,c))

print(paste(a,b,c, sep = "-"))

print(paste(a,b,c, sep = "", collapse = ""))
```

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

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

## Formatting numbers & strings - format() function

Numbers and strings can be formatted to a specific style using **format()** function.

### Syntax

The basic syntax for format function is –

```
format(x, digits, nsmall, scientific, width, justify = c("left", "right",
"centre", "none"))
```

Following is the description of the parameters used –

- **x** is the vector input.
- **digits** is the total number of digits displayed.
- **nsmall** is the minimum number of digits to the right of the decimal point.
- **scientific** is set to TRUE to display scientific notation.
- **width** indicates the minimum width to be displayed by padding blanks in the beginning.
- **justify** is the display of the string to left, right or center.

### Example

[Live Demo](#)

```
# Total number of digits displayed. Last digit rounded off.
result <- format(23.123456789, digits = 9)
print(result)

# Display numbers in scientific notation.
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)

# The minimum number of digits to the right of the decimal point.
result <- format(23.47, nsmall = 5)
print(result)

# Format treats everything as a string.
result <- format(6)
print(result)

# Numbers are padded with blank in the beginning for width.
result <- format(13.7, width = 6)
print(result)
```

```
# Left justify strings.
result <- format("Hello", width = 8, justify = "l")
print(result)

# Justfy string with center.
result <- format("Hello", width = 8, justify = "c")
print(result)
```

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

```
[1] "23.1234568"
[1] "6.000000e+00" "1.314521e+01"
[1] "23.47000"
[1] "6"
[1] "  13.7"
[1] "Hello  "
[1] " Hello  "
```

## Counting number of characters in a string - nchar() function

This function counts the number of characters including spaces in a string.

### Syntax

The basic syntax for nchar() function is –

```
nchar(x)
```

Following is the description of the parameters used –

- **x** is the vector input.

### Example

[Live Demo](#)

```
result <- nchar("Count the number of characters")
print(result)
```

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

```
[1] 30
```

## Changing the case - toupper() & tolower() functions

These functions change the case of characters of a string.

### Syntax

The basic syntax for toupper() & tolower() function is –

```
toupper(x)
tolower(x)
```

Following is the description of the parameters used –

- **x** is the vector input.

## Example

[Live Demo](#)

```
# Changing to Upper case.
result <- toupper("Changing To Upper")
print(result)

# Changing to lower case.
result <- tolower("Changing To Lower")
print(result)
```

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

```
[1] "CHANGING TO UPPER"
[1] "changing to lower"
```

## Extracting parts of a string - substring() function

This function extracts parts of a String.

### Syntax

The basic syntax for substring() function is –

```
substring(x, first, last)
```

Following is the description of the parameters used –

- **x** is the character vector input.
- **first** is the position of the first character to be extracted.
- **last** is the position of the last character to be extracted.

## Example

[Live Demo](#)

```
# Extract characters from 5th to 7th position.
result <- substring("Extract", 5, 7)
print(result)
```

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

```
[1] "act"
```

## R - Vectors

Vectors are the most basic R data objects and there are six types of atomic vectors. They are logical, integer, double, complex, character and raw.

### Vector Creation

#### Single Element Vector

Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

[Live Demo](#)



```
# Atomic vector of type character.
print("abc");

# Atomic vector of type double.
print(12.5)

# Atomic vector of type integer.
print(63L)

# Atomic vector of type logical.
print(TRUE)

# Atomic vector of type complex.
print(2+3i)

# Atomic vector of type raw.
print(charToRaw('hello'))
```

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

```
[1] "abc"
[1] 12.5
[1] 63
[1] TRUE
[1] 2+3i
[1] 68 65 6c 6c 6f
```

## Multiple Elements Vector

### Using colon operator with numeric data

[Live Demo](#)

```
# Creating a sequence from 5 to 13.
v <- 5:13
print(v)

# Creating a sequence from 6.6 to 12.6.
v <- 6.6:12.6
print(v)

# If the final element specified does not belong to the sequence then it is
discarded.
v <- 3.8:11.4
print(v)
```

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

```
[1] 5 6 7 8 9 10 11 12 13
[1] 6.6 7.6 8.6 9.6 10.6 11.6 12.6
[1] 3.8 4.8 5.8 6.8 7.8 8.8 9.8 10.8
```

### Using sequence (Seq.) operator

[Live Demo](#)

```
# Create vector with elements from 5 to 9 incrementing by 0.4.
print(seq(5, 9, by = 0.4))
```

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

```
[1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
```

## Using the c() function

The non-character values are coerced to character type if one of the elements is a character.

[Live Demo](#)

```
# The logical and numeric values are converted to characters.
s <- c('apple','red',5,TRUE)
print(s)
```

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

```
[1] "apple" "red"    "5"      "TRUE"
```

## Accessing Vector Elements

Elements of a Vector are accessed using indexing. The **[ ]** **brackets** are used for indexing. Indexing starts with position 1. Giving a negative value in the index drops that element from result. **TRUE**, **FALSE** or **0** and **1** can also be used for indexing.

[Live Demo](#)

```
# Accessing vector elements using position.
t <- c("Sun", "Mon", "Tue", "Wed", "Thurs", "Fri", "Sat")
u <- t[c(2,3,6)]
print(u)

# Accessing vector elements using logical indexing.
v <- t[c(TRUE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE)]
print(v)

# Accessing vector elements using negative indexing.
x <- t[c(-2,-5)]
print(x)

# Accessing vector elements using 0/1 indexing.
y <- t[c(0,0,0,0,0,0,1)]
print(y)
```

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

```
[1] "Mon" "Tue" "Fri"
[1] "Sun" "Fri"
[1] "Sun" "Tue" "Wed" "Fri" "Sat"
[1] "Sun"
```

## Vector Manipulation

### Vector arithmetic

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

[Live Demo](#)

```
# Create two vectors.
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11,0,8,1,2)

# Vector addition.
add.result <- v1+v2
print(add.result)

# Vector subtraction.
sub.result <- v1-v2
print(sub.result)

# Vector multiplication.
multi.result <- v1*v2
print(multi.result)

# Vector division.
divi.result <- v1/v2
print(divi.result)
```

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

```
[1]  7 19  4 13  1 13
[1] -1 -3  4 -3 -1  9
[1] 12 88  0 40  0 22
[1] 0.7500000 0.7272727      Inf 0.6250000 0.0000000 5.5000000
```

## Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

[Live Demo](#)

```
v1 <- c(3,8,4,5,0,11)
v2 <- c(4,11)
# V2 becomes c(4,11,4,11,4,11)

add.result <- v1+v2
print(add.result)

sub.result <- v1-v2
print(sub.result)
```

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

```
[1]  7 19  8 16  4 22
[1] -1 -3  0 -6 -4  0
```

## Vector Element Sorting

Elements in a vector can be sorted using the **sort()** function.

[Live Demo](#)

```
v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.
```

```

sort.result <- sort(v)
print(sort.result)

# Sort the elements in the reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
print(sort.result)

# Sorting character vectors in reverse order.
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)

```

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

```

[1] -9  0  3  4  5  8 11 304
[1] 304 11  8  5  4  3  0 -9
[1] "Blue" "Red" "violet" "yellow"
[1] "yellow" "violet" "Red" "Blue"

```

## R - Lists

Lists are the R objects which contain elements of different types like – numbers, strings, vectors and another list inside it. A list can also contain a matrix or a function as its elements. List is created using **list()** function.

### Creating a List

Following is an example to create a list containing strings, numbers, vectors and a logical values.

[Live Demo](#)

```

# Create a list containing strings, numbers, vectors and a logical
# values.
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)

```

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

```

[[1]]
[1] "Red"

[[2]]
[1] "Green"

[[3]]
[1] 21 32 11

[[4]]
[1] TRUE

[[5]]
[1] 51.23

[[6]]
[1] 119.1

```

## Naming List Elements

The list elements can be given names and they can be accessed using these names.

[Live Demo](#)

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan", "Feb", "Mar"), matrix(c(3, 9, 5, 1, -2, 8), nrow = 2),
  list("green", 12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Show the list.
print(list_data)
```

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

```
$`1st_Quarter`
[1] "Jan" "Feb" "Mar"

$A_Matrix
  [,1] [,2] [,3]
[1,]   3   5  -2
[2,]   9   1   8

$A_Inner_list
$A_Inner_list[[1]]
[1] "green"

$A_Inner_list[[2]]
[1] 12.3
```

## Accessing List Elements

Elements of the list can be accessed by the index of the element in the list. In case of named lists it can also be accessed using the names.

We continue to use the list in the above example –

[Live Demo](#)

```
# Create a list containing a vector, a matrix and a list.
list_data <- list(c("Jan", "Feb", "Mar"), matrix(c(3, 9, 5, 1, -2, 8), nrow = 2),
  list("green", 12.3))

# Give names to the elements in the list.
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")

# Access the first element of the list.
print(list_data[1])

# Access the thrid element. As it is also a list, all its elements will be
printed.
print(list_data[3])

# Access the list element using the name of the element.
print(list_data$A_Matrix)
```

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

```
$`1st_Quarter`  
[1] "Jan" "Feb" "Mar"  
  
$A_Inner_list  
$A_Inner_list[[1]]  
[1] "green"  
  
$A_Inner_list[[2]]  
[1] 12.3  
  
      [,1] [,2] [,3]  
[1,]    3    5   -2  
[2,]    9    1    8
```

## Manipulating List Elements

We can add, delete and update list elements as shown below. We can add and delete elements only at the end of a list. But we can update any element.

[Live Demo](#)

```
# Create a list containing a vector, a matrix and a list.  
list_data <- list(c("Jan","Feb","Mar"), matrix(c(3,9,5,1,-2,8), nrow = 2),  
  list("green",12.3))  
  
# Give names to the elements in the list.  
names(list_data) <- c("1st Quarter", "A_Matrix", "A Inner list")  
  
# Add element at the end of the list.  
list_data[4] <- "New element"  
print(list_data[4])  
  
# Remove the last element.  
list_data[4] <- NULL  
  
# Print the 4th Element.  
print(list_data[4])  
  
# Update the 3rd Element.  
list_data[3] <- "updated element"  
print(list_data[3])
```

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

```
[[1]]  
[1] "New element"  
  
$<NA>  
NULL  
  
$`A Inner list`  
[1] "updated element"
```

## Merging Lists

You can merge many lists into one list by placing all the lists inside one list() function.

```
# Create two lists.
list1 <- list(1,2,3)
list2 <- list("Sun", "Mon", "Tue")

# Merge the two lists.
merged.list <- c(list1,list2)

# Print the merged list.
print(merged.list)
```

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

```
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] "Sun"

[[5]]
[1] "Mon"

[[6]]
[1] "Tue"
```

## Converting List to Vector

A list can be converted to a vector so that the elements of the vector can be used for further manipulation. All the arithmetic operations on vectors can be applied after the list is converted into vectors. To do this conversion, we use the **unlist()** function. It takes the list as input and produces a vector.

```
# Create lists.
list1 <- list(1:5)
print(list1)

list2 <-list(10:14)
print(list2)

# Convert the lists to vectors.
v1 <- unlist(list1)
v2 <- unlist(list2)

print(v1)
print(v2)

# Now add the vectors
result <- v1+v2
```

```
print(result)
```

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

```
[[1]]  
[1] 1 2 3 4 5  
  
[[1]]  
[1] 10 11 12 13 14  
  
[1] 1 2 3 4 5  
[1] 10 11 12 13 14  
[1] 11 13 15 17 19
```

## R - Matrices

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. They contain elements of the same atomic types. Though we can create a matrix containing only characters or only logical values, they are not of much use. We use matrices containing numeric elements to be used in mathematical calculations.

A Matrix is created using the **matrix()** function.

### Syntax

The basic syntax for creating a matrix in R is –

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Following is the description of the parameters used –

- **data** is the input vector which becomes the data elements of the matrix.
- **nrow** is the number of rows to be created.
- **ncol** is the number of columns to be created.
- **byrow** is a logical clue. If TRUE then the input vector elements are arranged by row.
- **dimname** is the names assigned to the rows and columns.

### Example

Create a matrix taking a vector of numbers as input.

[Live Demo](#)

```
# Elements are arranged sequentially by row.  
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)  
print(M)  
  
# Elements are arranged sequentially by column.  
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)  
print(N)  
  
# Define the column and row names.  
rownames = c("row1", "row2", "row3", "row4")  
colnames = c("col1", "col2", "col3")  
  
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames,  
colnames))
```



```
print(P)
```

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

```
      [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
      [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
      col1 col2 col3
row1     3     4     5
row2     6     7     8
row3     9    10    11
row4    12    13    14
```

## Accessing Elements of a Matrix

Elements of a matrix can be accessed by using the column and row index of the element. We consider the matrix P above to find the specific elements below.

[Live Demo](#)

```
# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix.
P <- matrix(c(3:14), nrow = 4, byrow = TRUE, dimnames = list(rownames,
colnames))

# Access the element at 3rd column and 1st row.
print(P[1,3])

# Access the element at 2nd column and 4th row.
print(P[4,2])

# Access only the 2nd row.
print(P[2,])

# Access only the 3rd column.
print(P[,3])
```

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

```
[1] 5
[1] 13
col1 col2 col3
     6     7     8
row1 row2 row3 row4
     5     8    11    14
```

# Matrix Computations

Various mathematical operations are performed on the matrices using the R operators. The result of the operation is also a matrix.

The dimensions (number of rows and columns) should be same for the matrices involved in the operation.

## Matrix Addition & Subtraction

[Live Demo](#)

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)

# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)

# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

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

```
      [,1] [,2] [,3]
[1,]     3  -1    2
[2,]     9   4    6
      [,1] [,2] [,3]
[1,]     5   0    3
[2,]     2   9    4
Result of addition
      [,1] [,2] [,3]
[1,]     8  -1    5
[2,]    11  13   10
Result of subtraction
      [,1] [,2] [,3]
[1,]    -2  -1   -1
[2,]     7  -5    2
```

## Matrix Multiplication & Division

[Live Demo](#)

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print(matrix2)
```

```
# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

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

```
      [,1] [,2] [,3]
[1,]    3  -1    2
[2,]    9   4    6

      [,1] [,2] [,3]
[1,]    5   0    3
[2,]    2   9    4
Result of multiplication
      [,1] [,2] [,3]
[1,]   15   0    6
[2,]   18  36   24
Result of division
      [,1]      [,2]      [,3]
[1,]  0.6      -Inf  0.6666667
[2,]  4.5  0.4444444  1.5000000
```

## R - Arrays

Arrays are the R data objects which can store data in more than two dimensions. For example – If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.

An array is created using the **array()** function. It takes vectors as input and uses the values in the **dim** parameter to create an array.

### Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

[Live Demo](#)

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2))
print(result)
```

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

```
, , 1

      [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

```
, , 2
```

```
      [,1] [,2] [,3]  
[1,]     5    10    13  
[2,]     9    11    14  
[3,]     3    12    15
```

## Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the **dimnames** parameter.

[Live Demo](#)

```
# Create two vectors of different lengths.  
vector1 <- c(5,9,3)  
vector2 <- c(10,11,12,13,14,15)  
column.names <- c("COL1", "COL2", "COL3")  
row.names <- c("ROW1", "ROW2", "ROW3")  
matrix.names <- c("Matrix1", "Matrix2")  
  
# Take these vectors as input to the array.  
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames =  
list(row.names,column.names,  
      matrix.names))  
print(result)
```

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

```
, , Matrix1
```

```
      COL1 COL2 COL3  
ROW1     5    10    13  
ROW2     9    11    14  
ROW3     3    12    15
```

```
, , Matrix2
```

```
      COL1 COL2 COL3  
ROW1     5    10    13  
ROW2     9    11    14  
ROW3     3    12    15
```

## Accessing Array Elements

[Live Demo](#)

```
# Create two vectors of different lengths.  
vector1 <- c(5,9,3)  
vector2 <- c(10,11,12,13,14,15)  
column.names <- c("COL1", "COL2", "COL3")  
row.names <- c("ROW1", "ROW2", "ROW3")  
matrix.names <- c("Matrix1", "Matrix2")  
  
# Take these vectors as input to the array.  
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,  
      column.names, matrix.names))  
  
# Print the third row of the second matrix of the array.
```

```
print(result[3,,2])

# Print the element in the 1st row and 3rd column of the 1st matrix.
print(result[1,3,1])

# Print the 2nd Matrix.
print(result[, ,2])
```

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

```
COL1 COL2 COL3
  3   12   15
[1] 13
      COL1 COL2 COL3
ROW1    5   10   13
ROW2    9   11   14
ROW3    3   12   15
```

## Manipulating Array Elements

As array is made up matrices in multiple dimensions, the operations on elements of array are carried out by accessing elements of the matrices.

[Live Demo](#)

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
array1 <- array(c(vector1,vector2),dim = c(3,3,2))

# Create two vectors of different lengths.
vector3 <- c(9,1,0)
vector4 <- c(6,0,11,3,14,1,2,6,9)
array2 <- array(c(vector1,vector2),dim = c(3,3,2))

# create matrices from these arrays.
matrix1 <- array1[, ,2]
matrix2 <- array2[, ,2]

# Add the matrices.
result <- matrix1+matrix2
print(result)
```

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

```
 [,1] [,2] [,3]
[1,]  10  20  26
[2,]  18  22  28
[3,]   6  24  30
```

## Calculations Across Array Elements

We can do calculations across the elements in an array using the **apply()** function.

## Syntax

`apply(x, margin, fun)`

Following is the description of the parameters used –

- **x** is an array.
- **margin** is the name of the data set used.
- **fun** is the function to be applied across the elements of the array.

## Example

We use the `apply()` function below to calculate the sum of the elements in the rows of an array across all the matrices.

[Live Demo](#)

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)

# Take these vectors as input to the array.
new.array <- array(c(vector1,vector2),dim = c(3,3,2))
print(new.array)

# Use apply to calculate the sum of the rows across all the matrices.
result <- apply(new.array, c(1), sum)
print(result)
```

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

```
, , 1

      [,1] [,2] [,3]
[1,]     5    10    13
[2,]     9    11    14
[3,]     3    12    15

, , 2

      [,1] [,2] [,3]
[1,]     5    10    13
[2,]     9    11    14
[3,]     3    12    15

[1] 56 68 60
```

## R - Factors

Factors are the data objects which are used to categorize the data and store it as levels. They can store both strings and integers. They are useful in the columns which have a limited number of unique values. Like "Male", "Female" and True, False etc. They are useful in data analysis for statistical modeling.

Factors are created using the **factor ()** function by taking a vector as input.

## Example

[Live Demo](#)

```
# Create a vector as input.
data <-
c("East", "West", "East", "North", "North", "East", "West", "West", "West", "East", "North")

print(data)
print(is.factor(data))

# Apply the factor function.
factor_data <- factor(data)

print(factor_data)
print(is.factor(factor_data))
```

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

```
[1] "East" "West" "East" "North" "North" "East" "West" "West" "West" "East" "North"
"East" "North"
[1] FALSE
[1] East West East North North East West West West East North
Levels: East North West
[1] TRUE
```

## Factors in Data Frame

On creating any data frame with a column of text data, R treats the text column as categorical data and creates factors on it.

[Live Demo](#)

```
# Create the vectors for data frame.
height <- c(132,151,162,139,166,147,122)
weight <- c(48,49,66,53,67,52,40)
gender <- c("male","male","female","female","male","female","male")

# Create the data frame.
input_data <- data.frame(height,weight,gender)
print(input_data)

# Test if the gender column is a factor.
print(is.factor(input_data$gender))

# Print the gender column so see the levels.
print(input_data$gender)
```

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

```
  height weight gender
1    132    48   male
2    151    49   male
3    162    66 female
4    139    53 female
5    166    67   male
6    147    52 female
7    122    40   male
[1] TRUE
[1] male  male  female female male  female male
Levels: female male
```

## Changing the Order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

[Live Demo](#)

```
data <- c("East", "West", "East", "North", "North", "East", "West",  
         "West", "West", "East", "North")  
# Create the factors  
factor_data <- factor(data)  
print(factor_data)  
  
# Apply the factor function with required order of the level.  
new_order_data <- factor(factor_data, levels = c("East", "West", "North"))  
print(new_order_data)
```

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

```
[1] East  West  East  North North East  West  West  West  East  North  
Levels: East North West  
[1] East  West  East  North North East  West  West  West  East  North  
Levels: East West North
```

## Generating Factor Levels

We can generate factor levels by using the **gl()** function. It takes two integers as input which indicates how many levels and how many times each level.

### Syntax

```
gl(n, k, labels)
```

Following is the description of the parameters used –

- **n** is a integer giving the number of levels.
- **k** is a integer giving the number of replications.
- **labels** is a vector of labels for the resulting factor levels.

### Example

[Live Demo](#)

```
v <- gl(3, 4, labels = c("Tampa", "Seattle", "Boston"))  
print(v)
```

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

```
Tampa  Tampa  Tampa  Tampa  Seattle Seattle Seattle Seattle Boston  
[10] Boston  Boston  Boston  
Levels: Tampa Seattle Boston
```

## R - Data Frames

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

[Live Demo](#)

```
# 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.

[Live Demo](#)

```
# 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.

[Live Demo](#)

```
# 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.

[Live Demo](#)

```
# 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

[Live Demo](#)

```
# 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 3<sup>rd</sup> and 5<sup>th</sup> row with 2<sup>nd</sup> and 4<sup>th</sup> column

[Live Demo](#)

```
# 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.

[Live Demo](#)

```
# 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" column.
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.

[Live Demo](#)

```
# 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

## R - Packages

R packages are a collection of R functions, compiled code and sample data. They are stored under a directory called "**library**" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at [R Packages](#).

Below is a list of commands to be used to check, verify and use the R packages.

### Check Available R Packages

Get library locations containing R packages

[Live Demo](#)

```
.libPaths()
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

```
[2] "C:/Program Files/R/R-3.2.2/library"
```

### Get the list of all the packages installed

[Live Demo](#)

```
library()
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

```
Packages in library 'C:/Program Files/R/R-3.2.2/library':
```

```
base                      The R Base Package
```

boot	Bootstrap Functions (Originally by Angelo Canty for S)
class	Functions for Classification
cluster	"Finding Groups in Data": Cluster Analysis Extended Rousseeuw et al.
codetools	Code Analysis Tools for R
compiler	The R Compiler Package
datasets	The R Datasets Package
foreign	Read Data Stored by 'Minitab', 'S', 'SAS', 'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...
graphics	The R Graphics Package
grDevices	The R Graphics Devices and Support for Colours and Fonts
grid	The Grid Graphics Package
KernSmooth	Functions for Kernel Smoothing Supporting Wand & Jones (1995)
lattice	Trellis Graphics for R
MASS	Support Functions and Datasets for Venables and Ripley's MASS
Matrix	Sparse and Dense Matrix Classes and Methods
methods	Formal Methods and Classes
mgcv	Mixed GAM Computation Vehicle with GCV/AIC/REML Smoothness Estimation
nlme	Linear and Nonlinear Mixed Effects Models
nnet	Feed-Forward Neural Networks and Multinomial Log-Linear Models
parallel	Support for Parallel computation in R
rpart	Recursive Partitioning and Regression Trees
spatial	Functions for Kriging and Point Pattern Analysis
splines	Regression Spline Functions and Classes
stats	The R Stats Package
stats4	Statistical Functions using S4 Classes
survival	Survival Analysis
tcltk	Tcl/Tk Interface
tools	Tools for Package Development
utils	The R Utils Package

Get all packages currently loaded in the R environment

[Live Demo](#)

```
search()
```

When we execute the above code, it produces the following result. It may vary depending on the local settings of your pc.

```
[1] ".GlobalEnv"      "package:stats"    "package:graphics"
[4] "package:grDevices" "package:utils"    "package:datasets"
[7] "package:methods"  "Autoloads"        "package:base"
```

## Install a New Package

There are two ways to add new R packages. One is installing directly from the CRAN directory and another is downloading the package to your local system and installing it manually.

## Install directly from CRAN

The following command gets the packages directly from CRAN webpage and installs the package in the R environment. You may be prompted to choose a nearest mirror. Choose the one appropriate to your location.

```
install.packages("Package Name")

# Install the package named "XML".
install.packages("XML")
```

## Install package manually

Go to the link [R Packages](#) to download the package needed. Save the package as a **.zip** file in a suitable location in the local system.

Now you can run the following command to install this package in the R environment.

```
install.packages(file_name_with_path, repos = NULL, type = "source")

# Install the package named "XML"
install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type = "source")
```

## Load Package to Library

Before a package can be used in the code, it must be loaded to the current R environment. You also need to load a package that is already installed previously but not available in the current environment.

A package is loaded using the following command –

```
library("package Name", lib.loc = "path to library")

# Load the package named "XML"
install.packages("E:/XML_3.98-1.3.zip", repos = NULL, type = "source")
```

# R - Data Reshaping

Data Reshaping in R is about changing the way data is organized into rows and columns. Most of the time data processing in R is done by taking the input data as a data frame. It is easy to extract data from the rows and columns of a data frame but there are situations when we need the data frame in a format that is different from format in which we received it. R has many functions to split, merge and change the rows to columns and vice-versa in a data frame.

## Joining Columns and Rows in a Data Frame

We can join multiple vectors to create a data frame using the **cbind()** function. Also we can merge two data frames using **rbind()** function.

[Live Demo](#)

```
# Create vector objects.
city <- c("Tampa", "Seattle", "Hartford", "Denver")
state <- c("FL", "WA", "CT", "CO")
zipcode <- c(33602, 98104, 06161, 80294)

# Combine above three vectors into one data frame.
addresses <- cbind(city, state, zipcode)
```

```

# Print a header.
cat("# # # # The First data frame\n")

# Print the data frame.
print(addresses)

# Create another data frame with similar columns
new.address <- data.frame(
  city = c("Lowry", "Charlotte"),
  state = c("CO", "FL"),
  zipcode = c("80230", "33949"),
  stringsAsFactors = FALSE
)

# Print a header.
cat("# # # The Second data frame\n")

# Print the data frame.
print(new.address)

# Combine rows from both the data frames.
all.addresses <- rbind(addresses, new.address)

# Print a header.
cat("# # # The combined data frame\n")

# Print the result.
print(all.addresses)

```

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

```

# # # # The First data frame
      city      state zipcode
[1,] "Tampa"    "FL"   "33602"
[2,] "Seattle"  "WA"   "98104"
[3,] "Hartford" "CT"   "6161"
[4,] "Denver"   "CO"   "80294"

# # # The Second data frame
      city      state  zipcode
1      Lowry      CO    80230
2    Charlotte  FL     33949

# # # The combined data frame
      city      state  zipcode
1      Tampa      FL    33602
2    Seattle      WA    98104
3    Hartford      CT     6161
4      Denver      CO    80294
5      Lowry      CO    80230
6    Charlotte      FL    33949

```

## Merging Data Frames

We can merge two data frames by using the **merge()** function. The data frames must have same column names on which the merging happens.



In the example below, we consider the data sets about Diabetes in Pima Indian Women available in the library names "MASS". we merge the two data sets based on the values of blood pressure("bp") and body mass index("bmi"). On choosing these two columns for merging, the records where values of these two variables match in both data sets are combined together to form a single data frame.

[Live Demo](#)

```
library(MASS)
merged.Pima <- merge(x = Pima.te, y = Pima.tr,
  by.x = c("bp", "bmi"),
  by.y = c("bp", "bmi")
)
print(merged.Pima)
nrow(merged.Pima)
```

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

	bp	bmi	npreg.x	glu.x	skin.x	ped.x	age.x	type.x	npreg.y	glu.y	skin.y	ped.y
1	60	33.8	1	117	23	0.466	27	No	2	125	20	0.088
2	64	29.7	2	75	24	0.370	33	No	2	100	23	0.368
3	64	31.2	5	189	33	0.583	29	Yes	3	158	13	0.295
4	64	33.2	4	117	27	0.230	24	No	1	96	27	0.289
5	66	38.1	3	115	39	0.150	28	No	1	114	36	0.289
6	68	38.5	2	100	25	0.324	26	No	7	129	49	0.439
7	70	27.4	1	116	28	0.204	21	No	0	124	20	0.254
8	70	33.1	4	91	32	0.446	22	No	9	123	44	0.374
9	70	35.4	9	124	33	0.282	34	No	6	134	23	0.542
10	72	25.6	1	157	21	0.123	24	No	4	99	17	0.294
11	72	37.7	5	95	33	0.370	27	No	6	103	32	0.324
12	74	25.9	9	134	33	0.460	81	No	8	126	38	0.162
13	74	25.9	1	95	21	0.673	36	No	8	126	38	0.162
14	78	27.6	5	88	30	0.258	37	No	6	125	31	0.565
15	78	27.6	10	122	31	0.512	45	No	6	125	31	0.565
16	78	39.4	2	112	50	0.175	24	No	4	112	40	0.236
17	88	34.5	1	117	24	0.403	40	Yes	4	127	11	0.598

	age.y	type.y
1	31	No
2	21	No
3	24	No
4	21	No
5	21	No
6	43	Yes
7	36	Yes
8	40	No
9	29	Yes
10	28	No
11	55	No
12	39	No
13	39	No
14	49	Yes
15	49	Yes
16	38	No
17	28	No

[1] 17

# Melting and Casting

One of the most interesting aspects of R programming is about changing the shape of the data in multiple steps to get a desired shape. The functions used to do this are called **melt()** and **cast()**.

We consider the dataset called ships present in the library called "MASS".

[Live Demo](#)

```
library(MASS)
print(ships)
```

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

	type	year	period	service	incidents
1	A	60	60	127	0
2	A	60	75	63	0
3	A	65	60	1095	3
4	A	65	75	1095	4
5	A	70	60	1512	6
.....					
8	A	75	75	2244	11
9	B	60	60	44882	39
10	B	60	75	17176	29
11	B	65	60	28609	58
.....					
17	C	60	60	1179	1
18	C	60	75	552	1
19	C	65	60	781	0
.....					
.....					

## Melt the Data

Now we melt the data to organize it, converting all columns other than type and year into multiple rows.

```
molten.ships <- melt(ships, id = c("type", "year"))
print(molten.ships)
```

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

	type	year	variable	value
1	A	60	period	60
2	A	60	period	75
3	A	65	period	60
4	A	65	period	75
.....				
9	B	60	period	60
10	B	60	period	75
11	B	65	period	60
12	B	65	period	75
13	B	70	period	60
.....				
41	A	60	service	127
42	A	60	service	63

```

43      A    65    service    1095
.....
.....
70      D    70    service    1208
71      D    75    service      0
72      D    75    service    2051
73      E    60    service     45
74      E    60    service      0
75      E    65    service    789
.....
.....
101     C    70    incidents     6
102     C    70    incidents     2
103     C    75    incidents     0
104     C    75    incidents     1
105     D    60    incidents     0
106     D    60    incidents     0
.....
.....

```

## Cast the Molten Data

We can cast the molten data into a new form where the aggregate of each type of ship for each year is created. It is done using the **cast()** function.

```

recasted.ship <- cast(molten.ships, type+year~variable, sum)
print(recasted.ship)

```

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

```

      type year  period  service  incidents
1      A   60    135      190         0
2      A   65    135     2190         7
3      A   70    135     4865        24
4      A   75    135     2244        11
5      B   60    135    62058        68
6      B   65    135    48979       111
7      B   70    135    20163        56
8      B   75    135     7117        18
9      C   60    135     1731         2
10     C   65    135     1457         1
11     C   70    135     2731         8
12     C   75    135      274         1
13     D   60    135      356         0
14     D   65    135      480         0
15     D   70    135     1557        13
16     D   75    135     2051         4
17     E   60    135        45         0
18     E   65    135     1226        14
19     E   70    135     3318        17
20     E   75    135     542         1

```

## 1. Base R graphics

The `graphics` package is an R base package for creating graphs. The `plot` function is the [most basic function to create plots in R](#). With this plotting function you can create several types of plots, like line charts, barplots or even boxplots, depending on the input.

## 2. The ggplot2 package

The `ggplot2` package was developed by Hadley Wickham as an implementation of the "Grammar of Graphics". This way, charts are created using a layer structure that allows you to add more elements to the chart using a simple syntax.

## 3. The lattice package

The `lattice` library was created by Deepayan Sarkar and was developed as an alternative to the R base graphics tools, taking as an inspiration the Trellis graphics. This package is especially relevant when it comes to multivariate data representations.

# GRAPHICS IN R PROGRAMMING LANGUAGE

There are lots of types of plots in R and the choice will depend on the objectives of your study and on your data. Hence, you can create different types of statistical representations to discover all the patterns in the observations. These representations include:

## 1. Scatter plots

Scatter plots, also known as dispersion graphs, [scatter charts](#) or scatter diagrams allows to visually **check the relation between two or three variables** for 3D scatter plots. If you are working with several variables you can create a pairwise comparison with a scatter matrix.

## 2. Barplots

[Bar plots](#) are a type of graph very useful to represent the **count of any categorical variable**. You can also create bar charts for several groups or even summarize some characteristic of a variable depending against some groups. Other related graphs for categorical data in R are `spineplots` or `mosaicplots`.

### 3. Histograms

[Histograms](#) help to **understand the underlying distribution of the data** you are working with. This type of R graph can be easily created with the `hist` function.

### 4. Density plots

[Density plots](#) are **smoothed versions of the histograms** using non parametric statistical techniques.

### 5. Boxplots

[Box plots](#) are designed to **represent some characteristics of the data** like dispersion or skewness and also to **identify outliers**. The box represents the 50% of the data starting at the first quartile and ending in the third, where the central line represents the median.

### 6. Other plots

There are lots of graphics for representing data in addition to those previously mentioned, like dotcharts, qq-plots, beanplots among others.