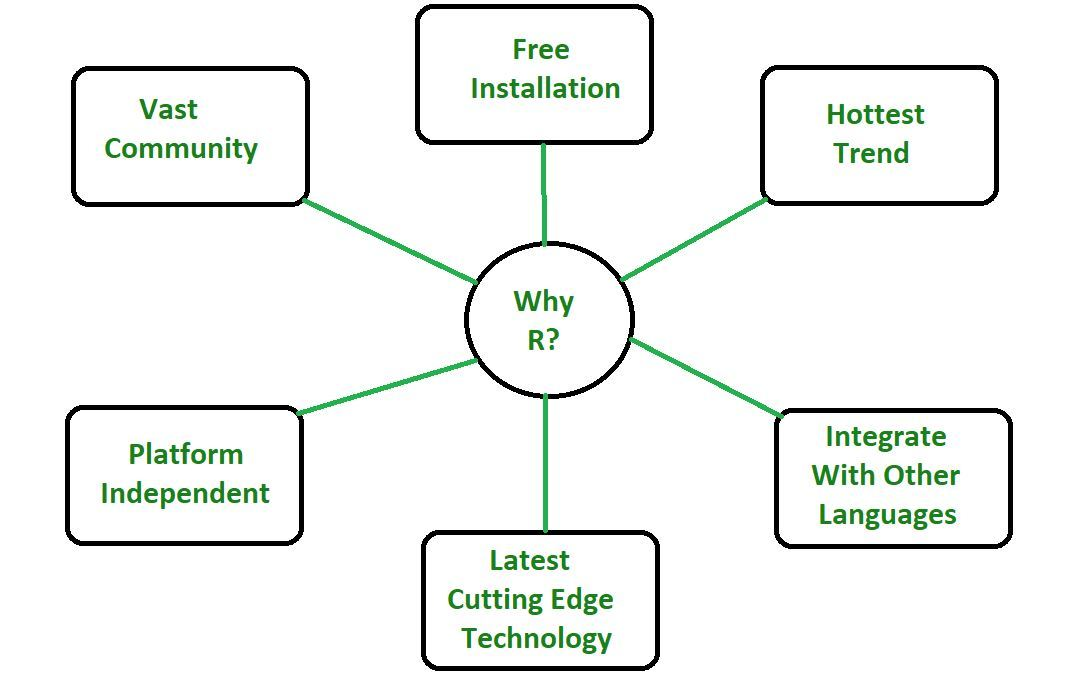
**R Programming Language – An Introduction**

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

It was designed by **Ross Ihaka and Robert Gentleman** at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project conceives in 1992, with an initial version released in 1995 and a stable beta version in 2000.

#### Why R Programming Language?



* R programming is used as a leading tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.
* It’s a platform-independent language. This means it can be applied to all operating system.
* It’s an open-source free language. That means anyone can install it in any organization without purchasing a license.
* R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.
* The R programming language has a vast community of users and it’s growing day by day.
* R is currently one of the most requested programming languages in the Data Science job market that makes it the hottest trend nowadays.

#### Features of R Programming Language

**Statistical Features of R:**

* **Basic Statistics:** The most common basic statistics terms are the mean, mode, and median. These are all known as “Measures of Central Tendency.” So using the R language we can measure central tendency very easily.
* **Static graphics:** R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.
* **Probability distributions:** Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distribution such as Binomial Distribution, Normal Distribution, Chi-squared Distribution and many more.

**Programming Features of R:**

* **R Packages:** One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.
* **Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages **ddR and multidplyr** used for distributed programming in R were released in November 2015.

#### Programming in R:

Since R is much similar to other widely used languages syntactically, it is easier to code and learn in R. Programs can be written in R in any of the widely used IDE like **R Studio, Rattle, Tinn-R**, etc. After writing the program save the file with the extension **.r**. To run the program use the following command on the command line:

R file\_name.r

# R program to print Welcome to GFG!

# Below line will print "Welcome to GFG!"

cat("Welcome to GFG!")

**Advantages of R:**

* R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.
* As R programming language is an open source. Thus, you can run R anywhere and at any time.
* R programming language is suitable for GNU/Linux and Windows operating system.
* R programming is cross-platform which runs on any operating system.
* In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

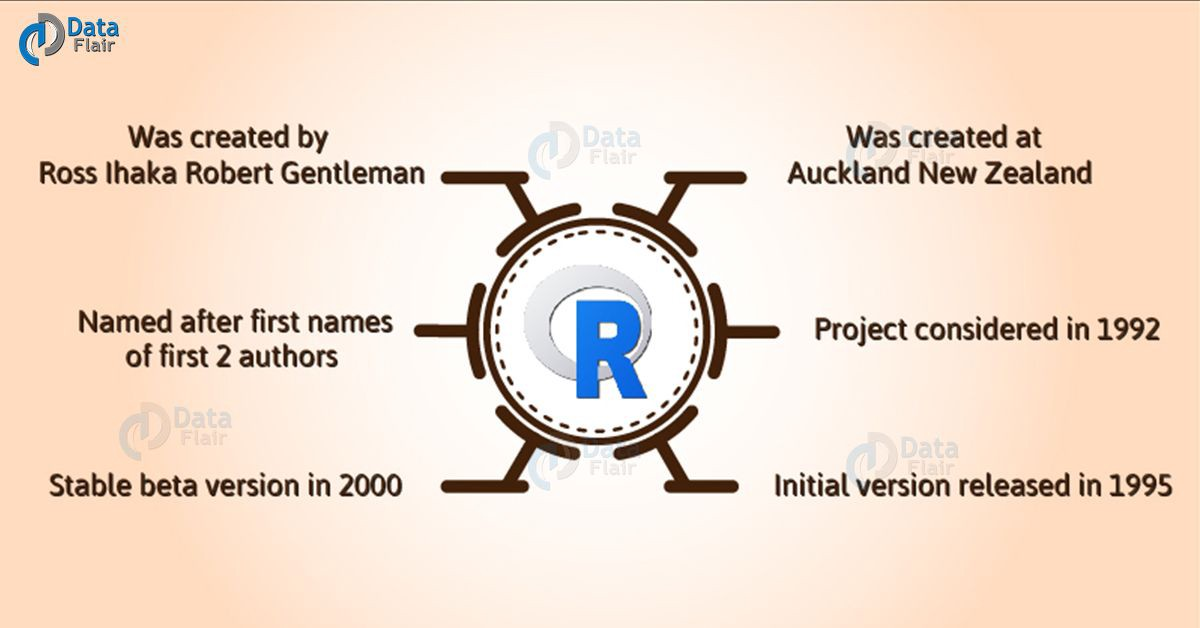
**Disadvantages of R:**

* In the R programming language, the standard of some packages is less than perfect.
* Although, R commands give little pressure to memory management. So R programming language may consume all available memory.
* In R basically, nobody to complain if something doesn’t work.

**Applications of R:**

* We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.
* R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.
* R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.
* Tech giants like Google, Facebook, bing, Accenture, Wipro and many more using R nowadays.

**History of R Programming**

****

R was **first implemented in the early 1990's by Robert Gentleman and Ross Ihaka**, both faculty members at the University of Auckland. Robert and Ross established R as an open source project in 1995. Since 1997, the R project has been managed by the R Core Group. And in February 2000, R 1.0.

# R vs Python

**R programming** and **Python**are both used extensively for Data Sciences. Both are very useful and open source languages as well.

**R Language** is used for machine learning algorithms, linear regression, time series, statistical inference, etc. It was designed by Ross Ihaka and Robert Gentleman in 1993.

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

**Python** is a widely-used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Below are some major differences between R and Python:

| Feature | R | Python |
| --- | --- | --- |
| **Introduction** | R is a language and environment for statistical programming which includes statistical computing and graphics. | Python is a general purpose programming language for data analysis and scientific computing |
| **Objective** | It has many features which are useful for statistical analysis and representation. | It can be used to develop GUI application and web application as well as with embedded systems |
| **Work ability** | It has many easy to use packages for performing tasks | It can easily perform matrix computation as well as optimization |
| **Integrated development environment** | Various popular R IDEs are Rstudio, RKward, R commander, etc. | Various popular Python IDEs are Spyder, Eclipse+Pydev, Atom, etc. |
| **Libraries and packages** | There are many packages and libraries like ggplot2, caret, etc. | Some essential packages and libraries are Pandas, Numpy, Scipy, etc. |
| **Scope** | It is mainly used for complex data analysis in data science. | It takes more streamline approach for data science projects. |

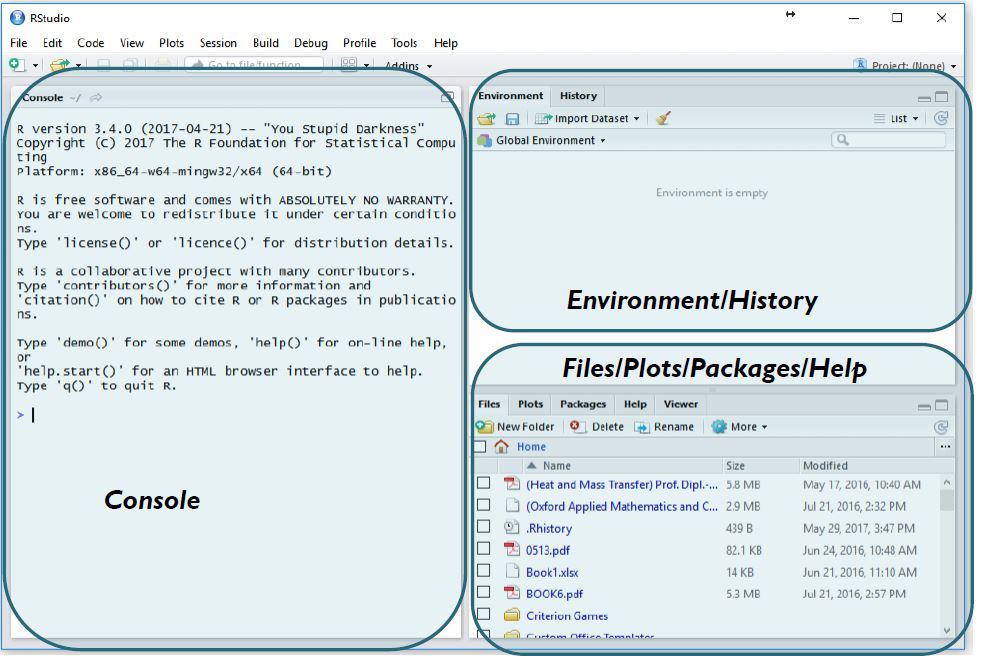
# Introduction to R Studio

R Studio is an integrated development environment(IDE) for R. IDE is a GUI, where you can write your quotes, see the results and also see the variables that are generated during the course of programming.

* R Studio is available as both Open source and Commercial software.
* R Studio is also available as both Desktop and Server version.
* R Studio is also available for various platforms such as Windows, Linux, and macOS.

R Studio can be downloaded from its official Website [rstudio.com](https://rstudio.com/products/rstudio/download/) and instructions for installation are available on **How to Install RStudio for R programming in Windows?**

After the Installation process is over, the R Studio Interface looks like:



* The console panel(left panel) is the place where R is waiting for you to tell it what to do, and see the results that are generated when you type in the commands.
* To the top right, you have Environmental/History panel. It contains 2 tabs:
  + **Environment tab:** It shows the variables that are generated during the course of programming in a workspace which is temporary.
  + **History tab:** In this tab, you’ll see all the commands that are used till now from the start of usage of R Studio.
* To the right bottom, you have another panel, which contains multiple tabs, such as files,  
  plots, packages, help, and viewer.
  + The **Files tab** shows the files and directories that are available within the default workspace of R.
  + The **Plots tab** shows the plots that are generated during the course of programming.
  + The **Packages tab** helps you to look at what are the packages that are already installed in the R Studio and it also gives a user interface to install new packages.
  + The **Help tab** is the most important one where you can get help from the R Documentation on the functions that are in built-in R.
  + The final and last tab is that the **Viewer tab** which can be used to see the local web content that’s generated using R.

### **Understanding Basic Data Types and Data Structures in R**

To make the best of the R language, we’ll need a strong understanding of the basic data types and data structures and how to operate on them.

Data structures are very important to understand because these are the objects you will manipulate on a day-to-day basis in R.

**Everything** in R is an object.

R has 6 basic data types.

* character
* numeric (real or decimal)
* integer
* logical
* complex

Elements of these data types may be combined to form data structures, such as atomic vectors. When we call a vector atomic, we mean that the vector only holds data of a single data type. Below are examples of atomic character vectors, numeric vectors, integer vectors, etc.

* **character**: "a", "swc"
* **numeric**: 2, 15.5
* **integer**: 2L (the L tells R to store this as an integer)
* **logical**: TRUE, FALSE
* **complex**: 1+4i (complex numbers with real and imaginary parts)

R provides many functions to examine features of vectors and other objects, for example

* class() - what kind of object is it (high-level)?
* typeof() - what is the object’s data type (low-level)?
* length() - how long is it? What about two dimensional objects?
* attributes() - does it have any metadata?

*# Example*

x <- "dataset"

typeof(x)

[1] "character"

attributes(x)

NULL

y <- 1:10

y

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

typeof(y)

[1] "integer"

length(y)

[1] 10

z <- as.numeric(y)

z

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

typeof(z)

[1] "double"

R has many **data structures**. These include

* atomic vector
* list
* matrix
* data frame
* factors

### **Vectors**

A vector is the most common and basic data structure in R and is pretty much the workhorse of R. Technically, vectors can be one of two types:

* atomic vectors
* lists

although the term “vector” most commonly refers to the atomic types not to lists.

### **The Different Vector Modes**

A vector is a collection of elements that are most commonly of mode character, logical, integer or numeric.

You can create an empty vector with vector(). (By default the mode is logical. You can be more explicit as shown in the examples below.) It is more common to use direct constructors such as character(), numeric(), etc.

vector() *# an empty 'logical' (the default) vector*

logical(0)

vector("character", length = 5) *# a vector of mode 'character' with 5 elements*

[1] "" "" "" "" ""

character(5) *# the same thing, but using the constructor directly*

[1] "" "" "" "" ""

numeric(5) *# a numeric vector with 5 elements*

[1] 0 0 0 0 0

logical(5) *# a logical vector with 5 elements*

[1] FALSE FALSE FALSE FALSE FALSE

You can also create vectors by directly specifying their content. R will then guess the appropriate mode of storage for the vector. For instance:

x <- c(1, 2, 3)

will create a vector x of mode numeric. These are the most common kind, and are treated as double precision real numbers. If you wanted to explicitly create integers, you need to add an L to each element (or coerce to the integer type using as.integer()).

x1 <- c(1L, 2L, 3L)

Using TRUE and FALSE will create a vector of mode logical:

y <- c(**TRUE**, **TRUE**, **FALSE**, **FALSE**)

While using quoted text will create a vector of mode character:

z <- c("Sarah", "Tracy", "Jon")

### **Examining Vectors**

The functions typeof(), length(), class() and str() provide useful information about your vectors and R objects in general.

typeof(z)

[1] "character"

length(z)

[1] 3

class(z)

[1] "character"

str(z)

chr [1:3] "Sarah" "Tracy" "Jon"

### **Adding Elements**

The function c() (for combine) can also be used to add elements to a vector.

z <- c(z, "Annette")

z

[1] "Sarah" "Tracy" "Jon" "Annette"

z <- c("Greg", z)

z

[1] "Greg" "Sarah" "Tracy" "Jon" "Annette"

### **Vectors from a Sequence of Numbers**

You can create vectors as a sequence of numbers.

series <- 1:10

seq(10)

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

seq(from = 1, to = 10, by = 0.1)

[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4

[16] 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9

[31] 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1 5.2 5.3 5.4

[46] 5.5 5.6 5.7 5.8 5.9 6.0 6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9

[61] 7.0 7.1 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.0 8.1 8.2 8.3 8.4

[76] 8.5 8.6 8.7 8.8 8.9 9.0 9.1 9.2 9.3 9.4 9.5 9.6 9.7 9.8 9.9

[91] 10.0

### **Missing Data**

R supports missing data in vectors. They are represented as NA (Not Available) and can be used for all the vector types covered in this lesson:

x <- c(0.5, **NA**, 0.7)

x <- c(**TRUE**, **FALSE**, **NA**)

x <- c("a", **NA**, "c", "d", "e")

x <- c(1+5i, 2-3i, **NA**)

The function is.na() indicates the elements of the vectors that represent missing data, and the function anyNA() returns TRUE if the vector contains any missing values:

x <- c("a", **NA**, "c", "d", **NA**)

y <- c("a", "b", "c", "d", "e")

is.na(x)

[1] FALSE TRUE FALSE FALSE TRUE

is.na(y)

[1] FALSE FALSE FALSE FALSE FALSE

anyNA(x)

[1] TRUE

anyNA(y)

[1] FALSE

### **Other Special Values**

Inf is infinity. You can have either positive or negative infinity.

1/0

[1] Inf

NaN means Not a Number. It’s an undefined value.

0/0

[1] NaN

### **What Happens When You Mix Types Inside a Vector?**

R will create a resulting vector with a mode that can most easily accommodate all the elements it contains. This conversion between modes of storage is called “coercion”. When R converts the mode of storage based on its content, it is referred to as “implicit coercion”. For instance, can you guess what the following do (without running them first)?

xx <- c(1.7, "a")

xx <- c(**TRUE**, 2)

xx <- c("a", **TRUE**)

You can also control how vectors are coerced explicitly using the as.<class\_name>() functions:

as.numeric("1")

[1] 1

as.character(1:2)

[1] "1" "2"

### **Objects Attributes**

Objects can have **attributes**. Attributes are part of the object. These include:

* names
* dimnames
* dim
* class
* attributes (contain metadata)

You can also glean other attribute-like information such as length (works on vectors and lists) or number of characters (for character strings).

length(1:10)

[1] 10

nchar("Software Carpentry")

[1] 18

### **Matrix**

In R matrices are an extension of the numeric or character vectors. They are not a separate type of object but simply an atomic vector with dimensions; the number of rows and columns. As with atomic vectors, the elements of a matrix must be of the same data type.

m <- matrix(nrow = 2, ncol = 2)

m

[,1] [,2]

[1,] NA NA

[2,] NA NA

dim(m)

[1] 2 2

You can check that matrices are vectors with a class attribute of matrix by using class() and typeof().

m <- matrix(c(1:3))

class(m)

[1] "matrix" "array"

typeof(m)

[1] "integer"

While class() shows that m is a matrix, typeof() shows that fundamentally the matrix is an integer vector.

## **Data types of matrix elements**

Consider the following matrix:

FOURS <- matrix(

c(4, 4, 4, 4),

nrow = 2,

ncol = 2)

Given that typeof(FOURS[1]) returns "double", what would you expect typeof(FOURS) to return? How do you know this is the case even without running this code?

Hint Can matrices be composed of elements of different data types?

## **Solution**

We know that typeof(FOURS) will also return "double" since matrices are made of elements of the same data type. Note that you could do something like as.character(FOURS) if you needed the elements of FOURS as characters.

Matrices in R are filled column-wise.

m <- matrix(1:6, nrow = 2, ncol = 3)

Other ways to construct a matrix

m <- 1:10

dim(m) <- c(2, 5)

This takes a vector and transforms it into a matrix with 2 rows and 5 columns.

Another way is to bind columns or rows using rbind() and cbind() (“row bind” and “column bind”, respectively).

x <- 1:3

y <- 10:12

cbind(x, y)

x y

[1,] 1 10

[2,] 2 11

[3,] 3 12

rbind(x, y)

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

x 1 2 3

y 10 11 12

You can also use the byrow argument to specify how the matrix is filled. From R’s own documentation:

mdat <- matrix(c(1, 2, 3, 11, 12, 13),

nrow = 2,

ncol = 3,

byrow = **TRUE**)

mdat

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

[1,] 1 2 3

[2,] 11 12 13

Elements of a matrix can be referenced by specifying the index along each dimension (e.g. “row” and “column”) in single square brackets.

mdat[2, 3]

[1] 13

### **List**

In R lists act as containers. Unlike atomic vectors, the contents of a list are not restricted to a single mode and can encompass any mixture of data types. Lists are sometimes called generic vectors, because the elements of a list can by of any type of R object, even lists containing further lists. This property makes them fundamentally different from atomic vectors.

A list is a special type of vector. Each element can be a different type.

Create lists using list() or coerce other objects using as.list(). An empty list of the required length can be created using vector()

x <- list(1, "a", **TRUE**, 1+4i)

x

[[1]]

[1] 1

[[2]]

[1] "a"

[[3]]

[1] TRUE

[[4]]

[1] 1+4i

x <- vector("list", length = 5) *# empty list*

length(x)

[1] 5

The content of elements of a list can be retrieved by using double square brackets.

x[[1]]

NULL

Vectors can be coerced to lists as follows:

x <- 1:10

x <- as.list(x)

length(x)

[1] 10

## **Examining Lists**

1. What is the class of x[1]?
2. What is the class of x[[1]]?

## **Solution**

Elements of a list can be named (i.e. lists can have the names attribute)

xlist <- list(a = "Karthik Ram", b = 1:10, data = head(mtcars))

xlist

$a

[1] "Karthik Ram"

$b

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

$data

mpg cyl disp hp drat wt qsec vs am gear carb

Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

names(xlist)

[1] "a" "b" "data"

## **Examining Named Lists**

1. What is the length of this object?
2. What is its structure?

## **Solution**

Lists can be extremely useful inside functions. Because the functions in R are able to return only a single object, you can “staple” together lots of different kinds of results into a single object that a function can return.

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

Elements are indexed by double brackets. Single brackets will still return a(nother) list. If the elements of a list are named, they can be referenced by the $ notation (i.e. xlist$data).

### **Data Frame**

A data frame is a very important data type in R. It’s pretty much the de facto data structure for most tabular data and what we use for statistics.

A data frame is a special type of list where every element of the list has same length (i.e. data frame is a “rectangular” list).

Data frames can have additional attributes such as rownames(), which can be useful for annotating data, like subject\_id or sample\_id. But most of the time they are not used.

Some additional information on data frames:

* Usually created by read.csv() and read.table(), i.e. when importing the data into R.
* Assuming all columns in a data frame are of same type, data frame can be converted to a matrix with data.matrix() (preferred) or as.matrix(). Otherwise type coercion will be enforced and the results may not always be what you expect.
* Can also create a new data frame with data.frame() function.
* Find the number of rows and columns with nrow(dat) and ncol(dat), respectively.
* Rownames are often automatically generated and look like 1, 2, …, n. Consistency in numbering of rownames may not be honored when rows are reshuffled or subset.

### **Creating Data Frames by Hand**

To create data frames by hand:

dat <- data.frame(id = letters[1:10], x = 1:10, y = 11:20)

dat

id x y

1 a 1 11

2 b 2 12

3 c 3 13

4 d 4 14

5 e 5 15

6 f 6 16

7 g 7 17

8 h 8 18

9 i 9 19

10 j 10 20

## **Useful Data Frame Functions**

* head() - shows first 6 rows
* tail() - shows last 6 rows
* dim() - returns the dimensions of data frame (i.e. number of rows and number of columns)
* nrow() - number of rows
* ncol() - number of columns
* str() - structure of data frame - name, type and preview of data in each column
* names() or colnames() - both show the names attribute for a data frame
* sapply(dataframe, class) - shows the class of each column in the data frame

See that it is actually a special list:

is.list(dat)

[1] TRUE

class(dat)

[1] "data.frame"

Because data frames are rectangular, elements of data frame can be referenced by specifying the row and the column index in single square brackets (similar to matrix).

dat[1, 3]

[1] 11

As data frames are also lists, it is possible to refer to columns (which are elements of such list) using the list notation, i.e. either double square brackets or a $.

dat[["y"]]

[1] 11 12 13 14 15 16 17 18 19 20

dat$y

[1] 11 12 13 14 15 16 17 18 19 20

The following table summarizes the one-dimensional and two-dimensional data structures in R in relation to diversity of data types they can contain.

| **Dimensions** | **Homogenous** | **Heterogeneous** |
| --- | --- | --- |
| 1-D | atomic vector | list |
| 2-D | matrix | data frame |

Lists can contain elements that are themselves muti-dimensional (e.g. a lists can contain data frames or another type of objects). Lists can also contain elements of any length, therefore list do not necessarily have to be “rectangular”. However in order for the list to qualify as a data frame, the length of each element has to be the same.

## **Column Types in Data Frames**

Knowing that data frames are lists, can columns be of different type?

What type of structure do you expect to see when you explore the structure of the PlantGrowth data frame? Hint: Use str().

## **Solution**

## **Key Points**

* R’s basic data types are character, numeric, integer, complex, and logical.
* R’s basic data structures include the vector, list, matrix, data frame, and factors. Some of these structures require that all members be of the same data type (e.g. vectors, matrices) while others permit multiple data types (e.g. lists, data frames).
* Objects may have attributes, such as name, dimension, and class.

## **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](http://tpcg.io/LcQaXR)

# 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](http://tpcg.io/pueuoc)

# 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](http://tpcg.io/tV7jUF)

# 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](http://tpcg.io/Kewzed)

# 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** functions gives the count of levels.

[Live Demo](http://tpcg.io/oKREZj)

# 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](http://tpcg.io/UiiDfw)

# 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