

## Unit – I

9 Hrs

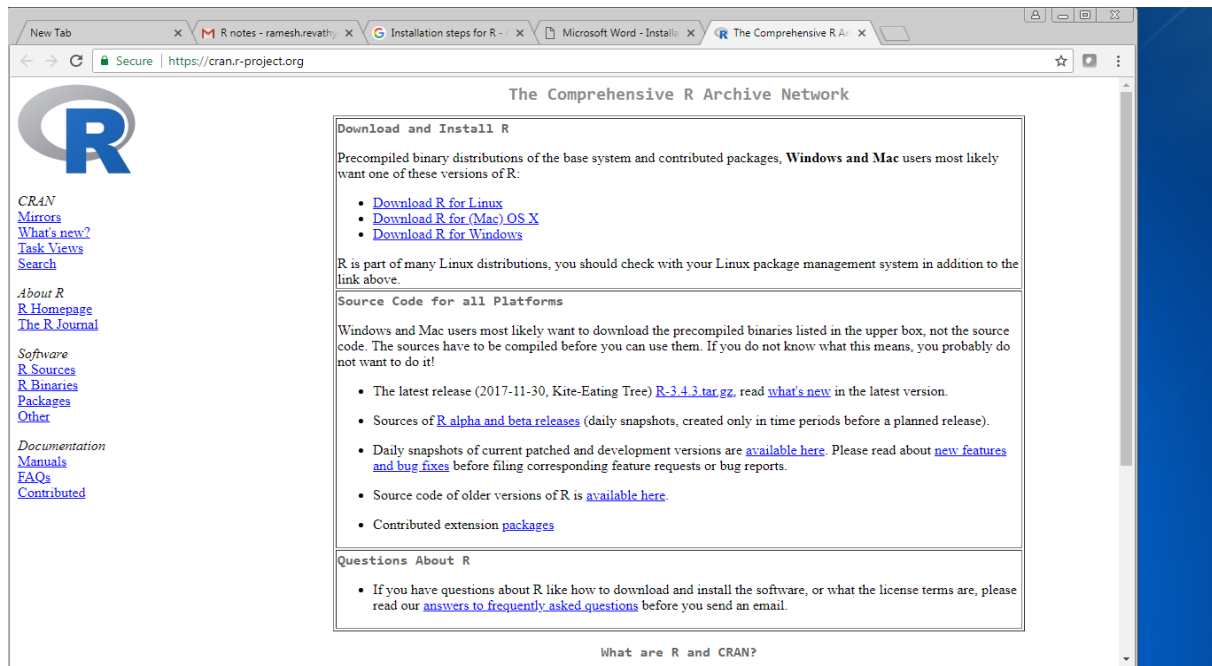
Introduction to R - History and fundamentals of R, Installation and use of R / R Studio / R Shiny, Installing R packages, R – Nuts and Bolts -Getting Data In and Out - Control Structures and Functions- Loop Functions- Data Manipulation- String Operations- Matrix Operations.

### History of R

- R is a programming language and free software environment for statistical computing and graphics that is supported by the R Foundation for Statistical Computing.
- R is an implementation of the S programming language combined with lexical scoping semantics inspired by Scheme.
- S was created by John Chambers in 1976, while at Bell Labs.
- There are some important differences, but much of the code written for S runs unaltered.<sup>[15]</sup>
- R was created by Ross Ihaka and Robert Gentleman<sup>[16]</sup> at the University of Auckland, New Zealand, and is currently developed by the *R Development Core Team*, of which Chambers is a member.
- R is named partly after the first names of the first two R authors and partly as a play on the name of S.
- The project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.

### Installation of R

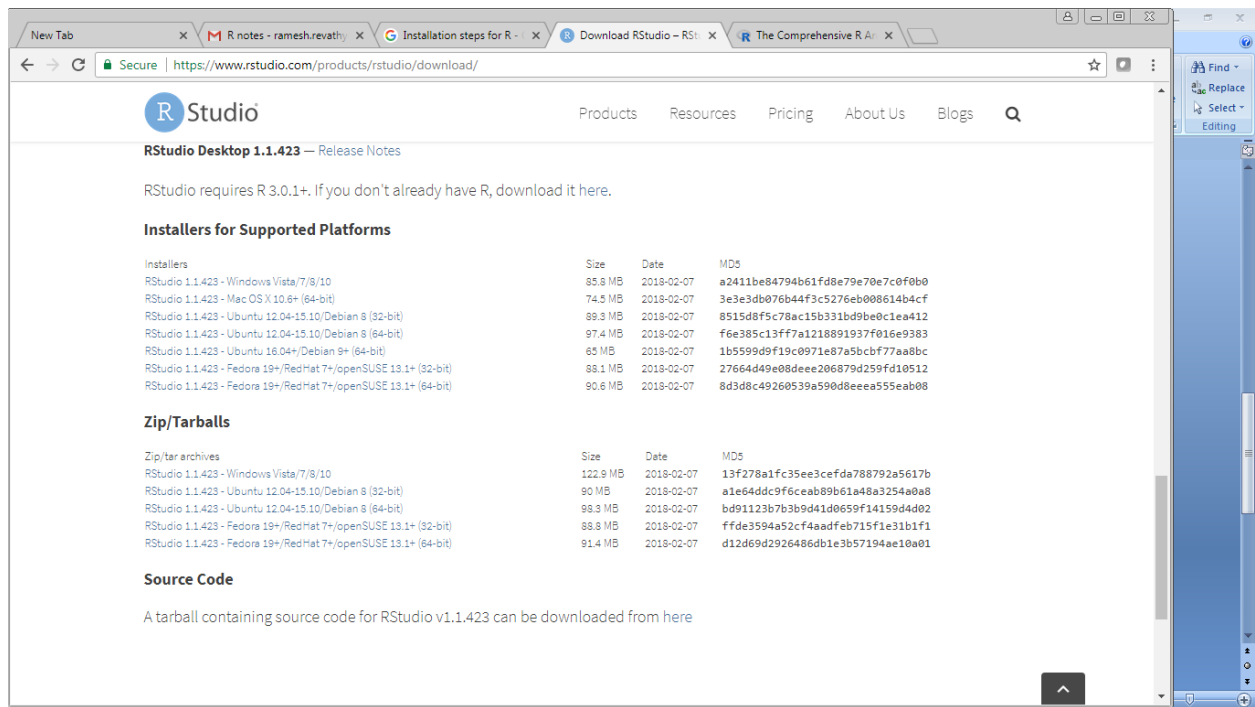
1. Download the R installer from <https://cran.r-project.org/>



2.Run the installer. Default settings are fine. If you do not have admin rights on your laptop, then ask you local IT support. In that case, it is important that you also ask them to give you full permissions to the R directories. Without this, you will not be able to install additional packages later.

## Installation of R Studio

1. Download RStudio: <https://www.rstudio.com/products/rstudio/download/>



- Once the installation of R has completed successfully, run the RStudio installer.
- If you do not have administrative rights on your laptop, step 2 may fail. Ask your IT Support or download a pre-built zip archive of RStudio which doesn't need installing.
- The link for this is towards the bottom of the download page, highlighted in Image
- Download the appropriate archive for your system (Windows/Linux only – the Mac version can be installed into your personal “Applications” folder without admin rights).
- Double clicking on the zip archive should automatically unpack it on most Windows machines.

## Installing R packages

There are three options to install packages.

### Option 1:

Click on the tab ‘ Packages’ then ‘Install’ as shown in figure 1.3. Click on Install to install R packages.

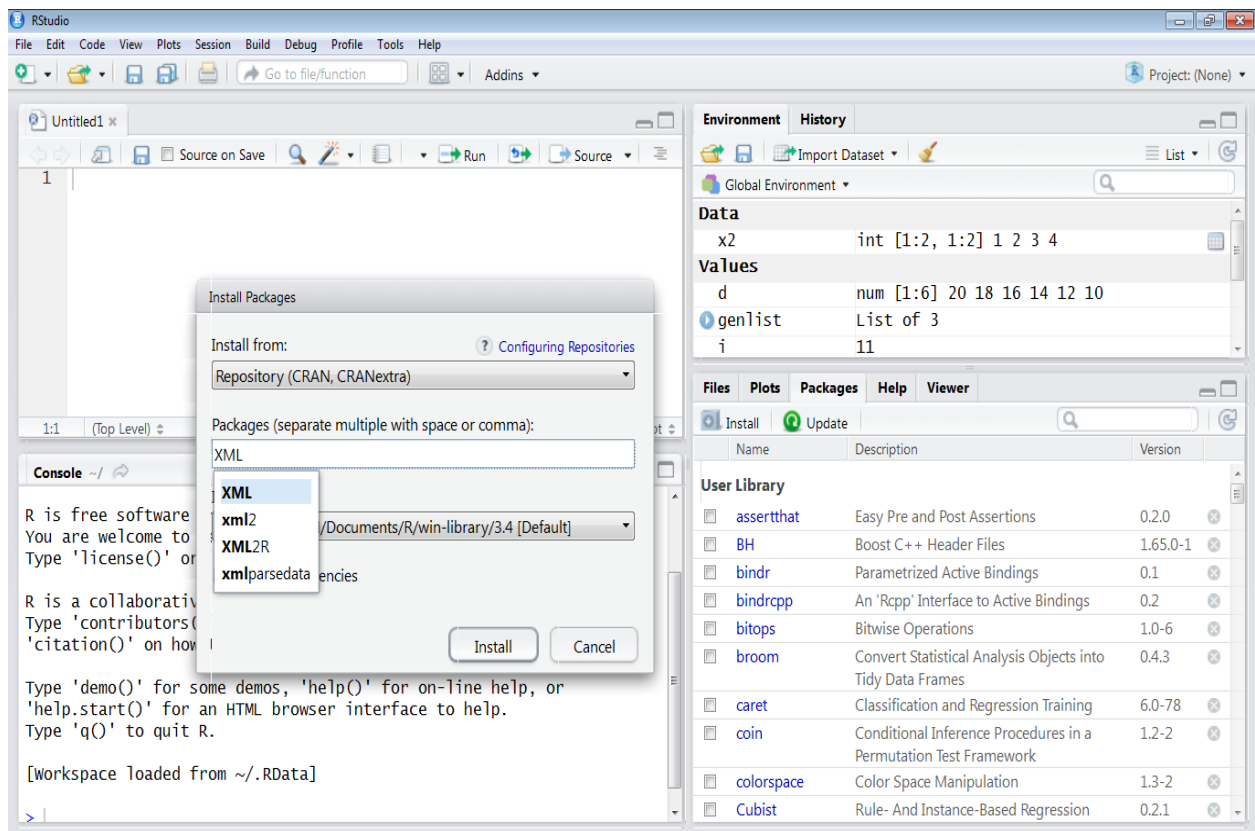


Fig 1.3

## Option 2:

Tools -> Install packages. As shown in Fig 1.4.

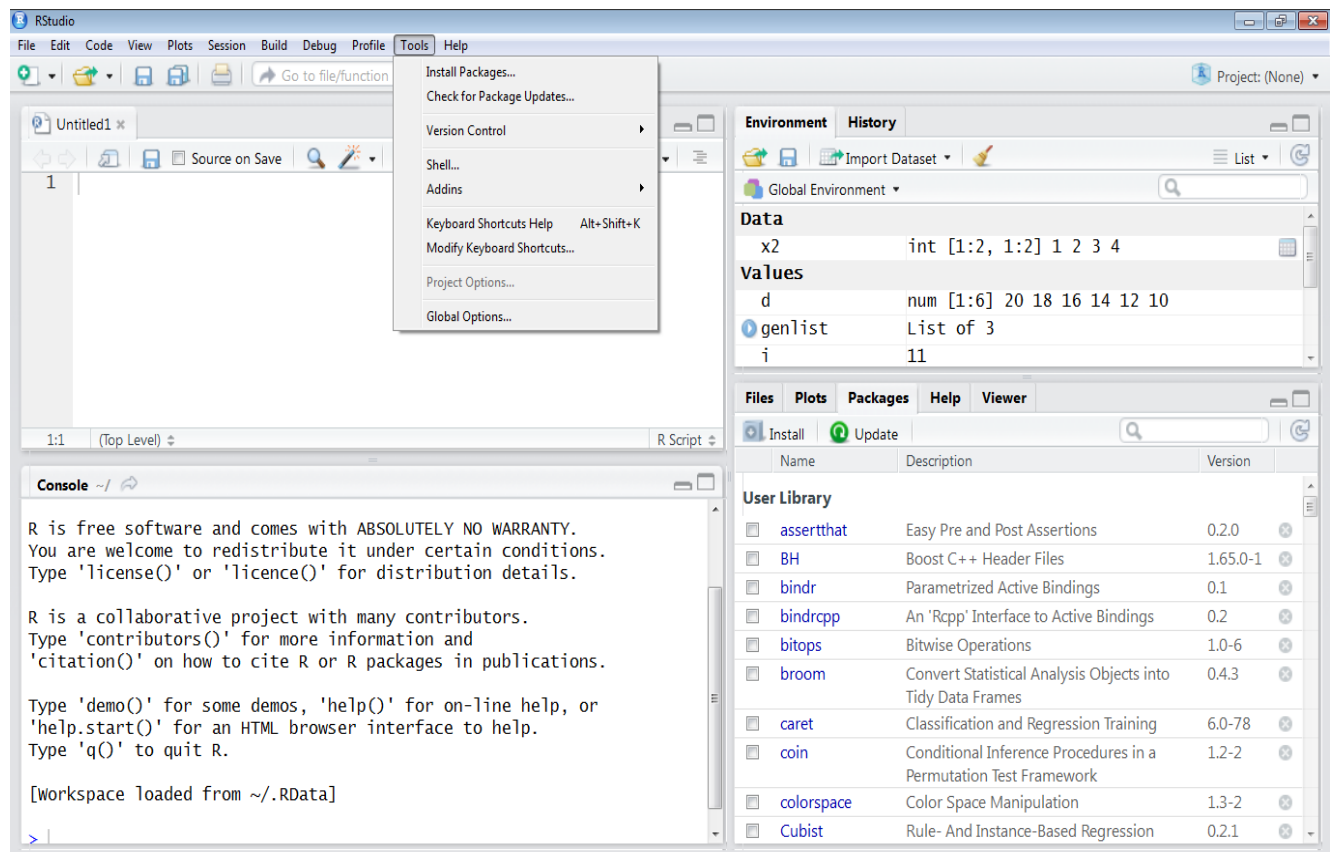
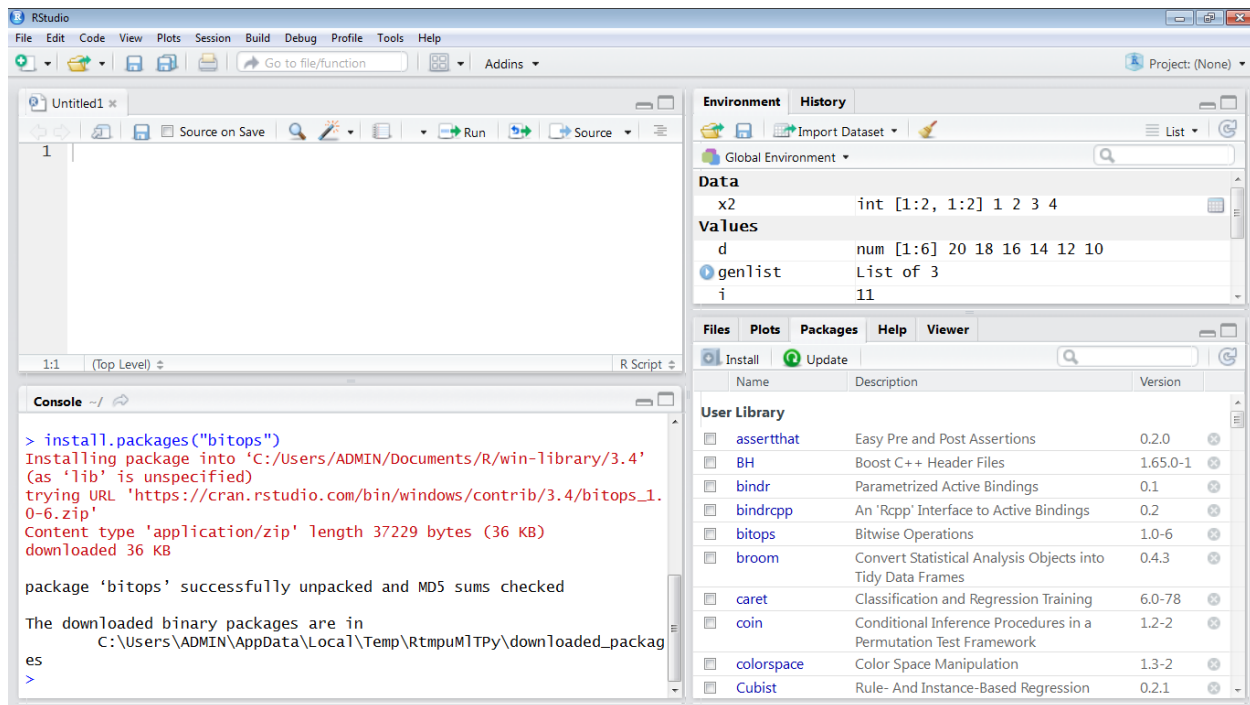


Fig 1.4

### Option 3:

Use the following command to install packages from R console.

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



**Note:** Check that the packages are installed by typing `'library(<packagename>')`

## Nuts and Bolts of R

- R is the most comprehensive statistical analysis package available. It incorporates all of the standard statistical tests, models, and analyses, as well as providing a comprehensive language for managing and manipulating data.
- R is a programming language and environment developed for statistical analysis by practising statisticians and researchers. It reflects well on a very competent community of computational statisticians.
- The graphical capabilities of R are outstanding, providing a fully programmable graphics language that surpasses most other statistical and graphical packages. ^
- R is free and open source software, allowing anyone to use and, importantly, to modify it. R is licensed under the GNU General Public License, with copyright held by The R Foundation for Statistical Computing.

- R has no license restrictions (other than ensuring our freedom to use it at our own discretion), and so we can run it anywhere and at any time.
- Anyone can provide new packages, and the wealth of quality packages available for R is a testament to this approach to software development and sharing.
- R has over 4800 packages available from multiple repositories specializing in topics like econometrics, data mining, spatial analysis, and bio-informatics.
- R is cross-platform. R runs on many operating systems and different hardware. It is popularly used on GNU/Linux, Macintosh, and Microsoft Windows, running on both 32 and 64 bit processors. ^
- R plays well with many other tools, importing data, for example, from CSV files, SAS, and SPSS, or directly from Microsoft Excel, Microsoft Access, Oracle, MySQL, and SQLite.
- It can also produce graphics output in PDF, JPG, PNG, and SVG formats, and table output for LATEX and HTML.
- R has active user groups .

## **Data In and Out**

There are a few principal functions reading data into R.

- `read.table()`, `read.csv()`: for reading tabular data
- `readLines()` : for reading lines of a text file
- `source ()` : for reading in R code files (inverse of `dump`)
- `dget ()` : for reading in R code files (inverse of `dput`)
- `load ()` : for reading in saved workspaces
- `unserialize ()` : for reading single R objects in binary form

There are analogous functions for writing data to files

- `write.table()` : for writing tabular data to text files (i.e. CSV) or connections
- `writeLines()` : for writing character data line-by-line to a file or connection
- `dump()` : for dumping a textual representation of multiple R objects
- `dput()` : for outputting a textual representation of an R object
- `save()` : for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- `Serialize()` : for converting an R object into a binary format for outputting to a connection (or file).

## Control Structures in R

Control Structures can be divided into three categories.

1. Conditional statements.

2. Looping Statements

3. Jump Statements

### Conditional Statements:

Conditional control 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**.

Example: If, If-Else, If-ElseIf Ladder, switch

#### **If:**

Description: An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax:     If(condition)

{



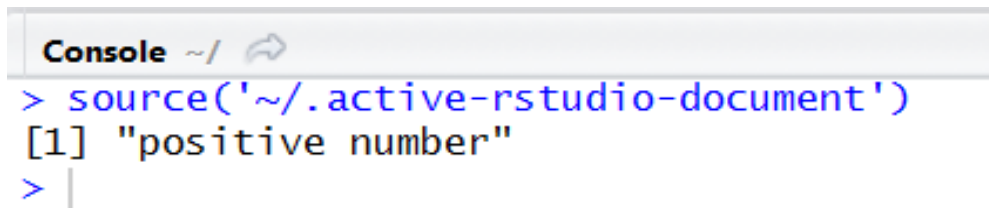
Statement

}

Example: Check for Positive Number

```
1 a=10
2 if(a>0)
3 {
4     print("positive number")
5 }
```

Output :



```
Console ~/
> source('~/.active-rstudio-document')
[1] "positive number"
> |
```

## If-Else

Description: An **if...else** statement contains the same elements as an if statement and then some extra:

- The keyword else, placed after the first code block
- Second block of code, contained within braces, that has to be carried out if and only if the result of the condition in the if() statement is FALSE

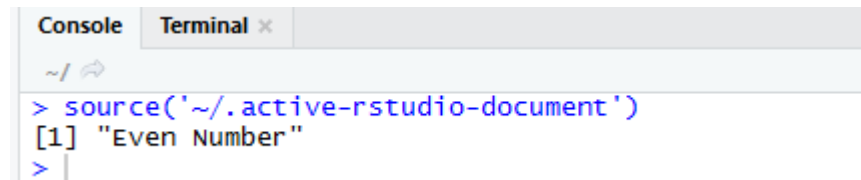
Syntax: if(condition)

```
{
    Statement1
}else{
    Statement 2
}
```

Example: Check for Odd or Even Number

```
1 a=10
2 if(a%%2==0)
3 {
4   print("Even Number")
5 }else{
6   print("Odd Number")
7 }
```

Output:



```
Console Terminal x
~/
> source('~/.active-rstudio-document')
[1] "Even Number"
>
```

## If-Else if Ladder

If-Else statements can be chained using If-Else if Ladder

Syntax:if(condition)

```
{
    Statement1
}else if{
    Statement 2
}else{
    Statement 3
}
```

Example:Greatest of three numbers

```

1 a=10
2 b=50
3 c=30
4 if((a>b)&&(a>c))
5 {
6   cat(a,"is greatest")
7 }else if(b>c){
8   cat(b,"is greatest")
9 }else{
10  cat(c,"is greatest")
11 }
12

```

Output:

```

Console Terminal x
~/
> source('~/.active-rstudio-document')
50 is greatest
> |

```

## Switch:

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax: `switch(Expression, "Option 1", "Option 2", "Option 3"....., "Option N")`

Example: Print Department names using Switch

```

switch.R x
1 x=switch(2,"CSE","IT","ECE")
2 print(x)

```

Output:

```
Console Terminal x
~/
> source('~switch.R')
[1] "IT"
> |
```

## Looping Statements in R

There may be a situation when you need to execute a block of code several number of times. A loop statement allows us to execute a statement or group of statements multiple times.

Example:for,while,Repeat

### For:

Description: A **For loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax: for(value in vector)

```
{
    Statements
}
```

Example:Printing elements of vector

```
for.R x
Source on Save
1 r=c(6,5,8,3)
2 for(i in r)
3   print(i)
```

Output:

```
Console Terminal x
~/
> source('~/.switch.R')
[1] "IT"
> source('~/.active-rstudio-document')
[1] 6
[1] 5
[1] 8
[1] 3
> |
```

## While:

Description: The While loop executes the same code again and again until a stop condition is met.

Syntax: while(condition){

Statement

}

Example: Sum of Series

```
Untitled1* x
Source on Save
1 i=1
2 sum=0
3 while(i<=10)
4 {
5     sum=sum+i
6     i=i+1
7 }
8 cat("Sum of first 10 numbers",sum)|
```

## Output:

```
Console Terminal x
~/
> source('~/.active-rstudio-document')
sum of first 10 numbers 55
> |
```

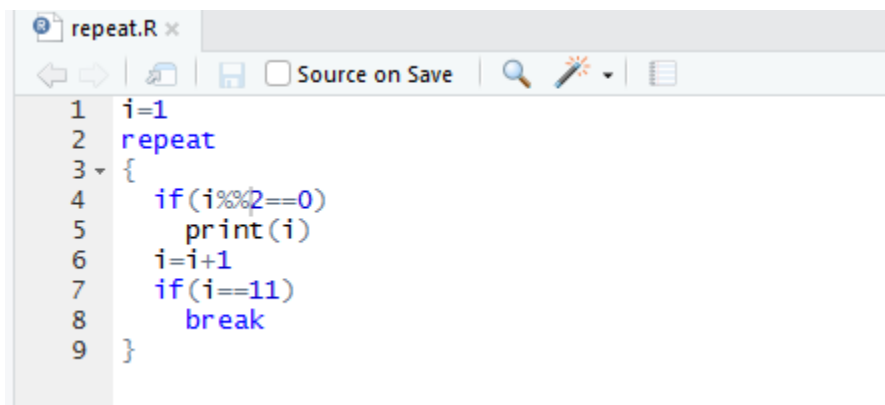
## Repeat:

Description: The **Repeat loop** executes the same code again and again until a stop condition is met.

Syntax: repeat

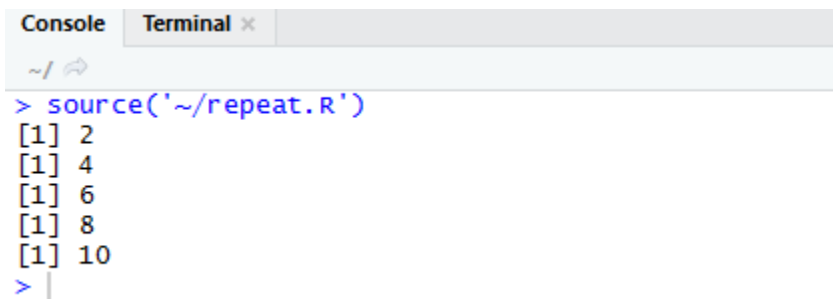
```
{  
    Commands  
    If(condition)  
    Break  
}
```

Example: Printing even numbers till 10



```
repeat.R x  
1 i=1  
2 repeat  
3 {  
4   if(i%%2==0)  
5     print(i)  
6   i=i+1  
7   if(i==11)  
8     break  
9 }
```

Output:



```
Console Terminal x  
~/  
> source('~/.repeat.R')  
[1] 2  
[1] 4  
[1] 6  
[1] 8  
[1] 10  
> |
```

## Jump Statements:

Description: Loop control statements change execution from its normal sequence.

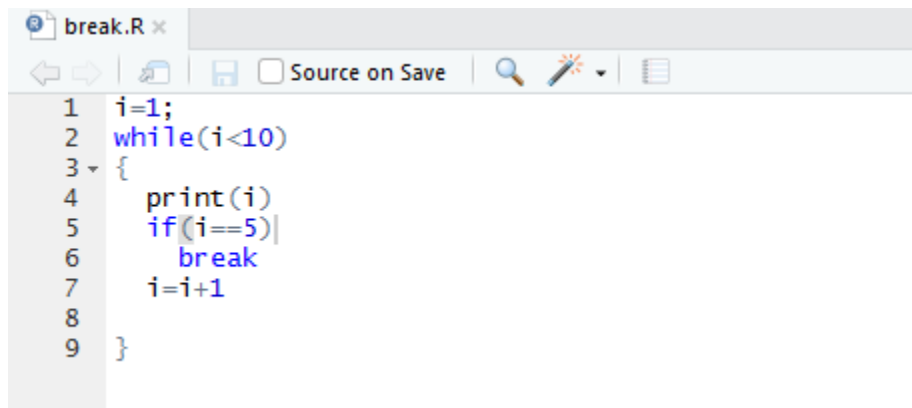
Example:break,next

### Break:

Terminates the **loop** statement and transfers execution to the statement immediately following the loop.

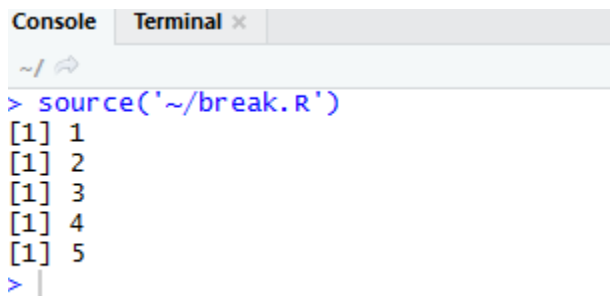
Syntax:break

Example:print numbers till 5



```
break.R x
1 i=1;
2 while(i<10)
3 {
4   print(i)
5   if(i==5){
6     break
7   }
8   i=i+1
9 }
```

Output:



```
Console Terminal x
~/
> source('~/.break.R')
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
> |
```

### Next:

Description:The **next** statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax:

next

Example:Print numbers from 1 to 10 except 5

```
next.R x
Source on Save
1 i=0;
2 while(i<10)
3 {
4   i=i+1
5   if(i==5)
6     next
7   print(i)
8
9 }
```

Output:

```
Console Terminal x
~/
> source('~'/next.R')
[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
> |
```

## Functions in R

Definition:

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.

Syntax:

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

Function name=function(ar1,ar2,.....)

{

Function body

}



## Function Components:

The different parts of a function are –

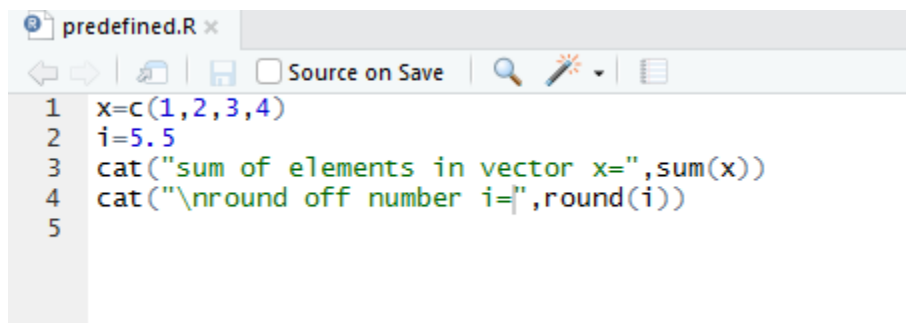
- **Function Name** – This is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments** – An argument is a placeholder. When a function is invoked, you pass a value to the argument. Arguments are optional; that is, a function may contain no arguments. 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.

## Built-in Functions

R has many **in-built** functions which can be directly called in the program without defining them first.

Eg: sum, seq, abs, round etc.

Example:



```
predefined.R x
1 x=c(1,2,3,4)
2 i=5.5
3 cat("sum of elements in vector x=",sum(x))
4 cat("\nround off number i=",round(i))
5
```

Output:

```

Console Terminal x
~/
> source('~/.active-rstudio-document')
sum of elements in vector x 10
round off number i 6
> |

```

## 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.

Function without Arguments:

Here the function does not receive any arguments.

Example:

```

function without arguments.R x
Source on Save Run
1 factorial=function()
2 {
3   fact=1
4   x=as.integer(readline("Enter the Number"))
5   for(i in 1:x)
6     fact=fact*i
7   cat("the Factorial of the given number is",fact)
8 }
9 factorial()

```

Output:

```

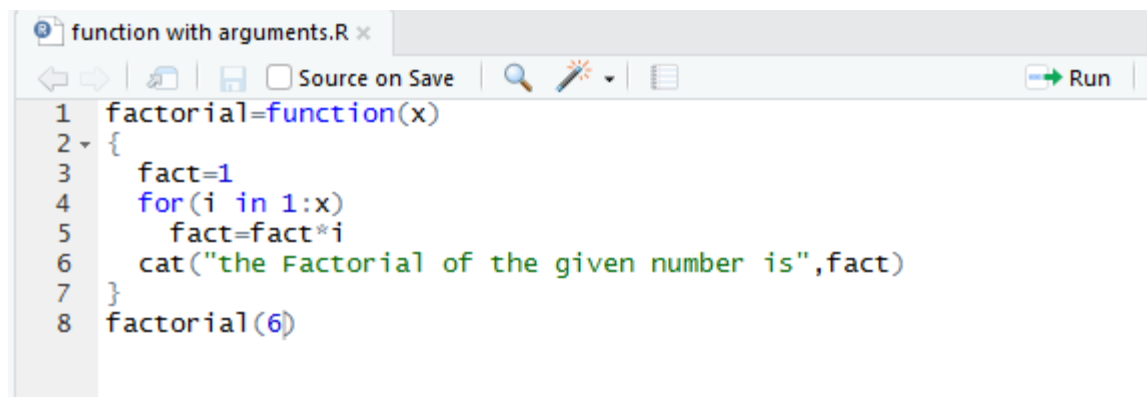
Console Terminal x
~/
> source('~/.function without arguments.R')
Enter the Number 6
the Factorial of the given number is 720
> |

```

## Function with Arguments:

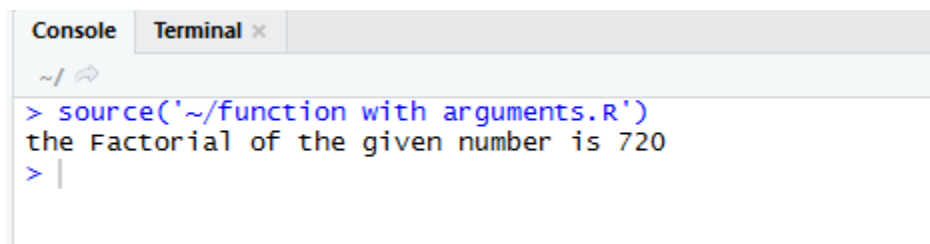
The arguments to a function call can be supplied in the same sequence as defined in the function or they can be supplied in a different sequence but assigned to the names of the arguments.

Example:



```
1 factorial=function(x)
2 {
3   fact=1
4   for(i in 1:x)
5     fact=fact*i
6   cat("the Factorial of the given number is",fact)
7 }
8 factorial(6)
```

Output:



```
> source('~/.function with arguments.R')
the Factorial of the given number is 720
> |
```

## 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.

Example:

```
Function with default values.R x
Source on Save Run
1 add=function(a=7,b=8)
2 {
3   return(a+b)
4 }
5 cat("Function call using default values",add())
6 cat("\nFunction call by specifying values",add(3,4))
7
```

Output:

```
Console Terminal x
~/
> source('~/.Function with default values.R')
Function call using default values 15
Function call by specifying values 7
> |
```

## LOOPING FUNCTIONS IN R

The for, while loops can often be replaced by looping functions:

### **lapply:**

Definition: Loop over a list and evaluate a function on each element

Syntax : lapply(X, FUN, ...)

X =List

FUN=A Function

.... =other arguments

Example:

```

Console Terminal x
~/
> v=c("Priya","Saadhana","Ramesh")
> lapply(v,nchar)
[[1]]
[1] 5

[[2]]
[1] 8

[[3]]
[1] 6

```

### sapply:

Definition : same as lapply but try to simplify the result .

Syntax : lapply(X, FUN, ...)

X =List

FUN=A Function

.... =other arguments

Example:

```

Console Terminal x
~/
> v=c("Priya","Saadhana","Ramesh")
> sapply(v,nchar)
  Priya Saadhana  Ramesh
    5         8      6
> |

```

### apply:

Definition: apply a function over the margins of an array

Syntax: apply(X, MARGIN, FUN, ...)

X =An Array

MARGIN =Integer vector indicating which margins should be "retained".

FUN =Function to be applied

... = other arguments to be passed to FUN

Example: Rowwise sum in a given matrix using apply()

```
Console Terminal x
~/
> x=matrix(1:6,3,2)
> x
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(x,1,sum)
[1] 5 7 9
> |
```

### **mapply:**

Definition: Multivariate version of lapply

Syntax: mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

FUN : the function to be applied

. . : arguments to apply over

MoreArgs : a list of other arguments to FUN

SIMPLIFY : logical; whether the result should be simplified to a vector or matrix.

Example:

```
Console Terminal x
~/
> mapply(rep,1:4,4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4

> |
```

## Matrices in R

Matrices are the R objects in which the elements are arranged in a two-dimensional rectangular layout. We use matrices containing numeric elements to be used in mathematical calculations.

### Syntax:

The basic syntax for creating a matrix in R is –


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

- **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.

### Matrix Creation:

(i) Arrange elements sequentially by row.

Example:


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

[Workspace loaded from ~/.RData]

> M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
>
> M
      [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
```

---


(ii) Arrange elements sequentially by column.

```
Console ~/ 
> M <- matrix(c(3:14), nrow = 4, byrow = FALSE)
>
> M
      [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
> |
```

### Accessing Elements of Matrix:

Elements of a matrix can be accessed by using the column and row index of the element

Example:


```
Console ~/ 
> M
      [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
> M[1][1]
[1] 3
> M[2][1]
[1] 4
> M[3][1]
[1] 5
> |
```

### Matrix Operations:

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:

```
Console ~/ 
> M <- matrix(c(1:9), nrow = 3, ncol=3, byrow = TRUE)
>
> M
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N <- matrix(c(11:19), nrow = 3, ncol=3, byrow = TRUE)
>
> N
      [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M+N
      [,1] [,2] [,3]
[1,]   12   14   16
[2,]   18   20   22
[3,]   24   26   28
> |
```

## Matrix Subtraction

```

Console ~/ 
> M
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
      [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M-N
      [,1] [,2] [,3]
[1,]   -10   -10   -10
[2,]   -10   -10   -10
[3,]   -10   -10   -10
> |

```

## Matrix Multiplication(Elementwise)

```

Console ~/ 
> M
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
      [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M*N
      [,1] [,2] [,3]
[1,]   11   24   39
[2,]   56   75   96
[3,]  119  144  171
> |

```

## Matrix Multiplication(Real)

```

Console ~/
> M
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
      [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M*%N
      [,1] [,2] [,3]
[1,]    90    96   102
[2,]   216   231   246
[3,]   342   366   390
> |

```

Matrix Division:

```

Console ~/
> M
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> N
      [,1] [,2] [,3]
[1,]   11   12   13
[2,]   14   15   16
[3,]   17   18   19
> M/N
      [,1]      [,2]      [,3]
[1,] 0.09090909 0.1666667 0.2307692
[2,] 0.28571429 0.3333333 0.3750000
[3,] 0.41176471 0.4444444 0.4736842
> |

```

## String Operations in R

### String:

Any value written within a pair of single quote or double quotes in R is treated as a string.

Example: `S="Hello"`

`S1='hai'`

### 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:

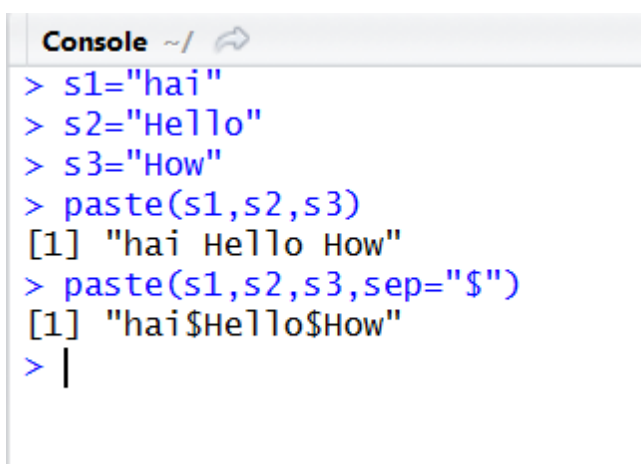
`paste(..., sep = " ", collapse = NULL)`


`...` 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



```
Console ~/ 
> s1="hai"
> s2="Hello"
> s3="How"
> paste(s1,s2,s3)
[1] "hai Hello How"
> paste(s1,s2,s3,sep="$")
[1] "hai$Hello$How"
> |
```

#### 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:

```
Console ~/   
> s1  
[1] "hai"  
> s2  
[1] "Hello"  
> nchar(s1)  
[1] 3  
> nchar(s2)  
[1] 5  
> |
```

## 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:

```
Console ~/   
> s1  
[1] "hai"  
> toupper(s1)  
[1] "HAI"  
> s2="HELLO"  
> tolower(s2)  
[1] "hello"  
> |
```

## 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:

```
Console ~/   
> s2  
[1] "HELLO"  
> substring(s2,1,3)  
[1] "HEL"  
> |
```

## Replacement Functions:sub() and gsub()


These are replacement functions, which replaces the occurrence of a substring with other substring.

- sub() Function in R replaces the first instance of a substring
- gsub() function in R replaces all the instances of a substring

### Syntax for sub() and gsub() function in R:

1. sub(old, new, string)
2. gsub(old, new, string)

### Example:

```
Console ~/   
> s2  
[1] "HELLO"  
> sub('L','R',s2)  
[1] "HERLO"  
> gsub('L','R',s2)  
[1] "HERRO"  
> |
```


### Pattern Matching Function: Grep() function

#### Syntax:

grep(value = FALSE) returns an integer vector of the indices of the elements of x that yielded a match .

grep(value = TRUE) returns a character vector containing the selected elements of x.

### Example:

```
Console ~/   
> str <- c("Regular", "expression", "examples of R language")  
> x <- grep("ex",str,value=F)  
> x  
[1] 2 3  
> x1 <- grep("ex",str,value=T)  
> x1  
[1] "expression"          "examples of R language"
```

