

R Programming

A Summary of Basic Skills with Examples

Prepared for MSc Big Data Analytics Students

By

Aliyu Sambo

1/25/2021

Table of Contents

Introduction	3
Mathematical Operators and Functions	3
Logical Operators	4
Working with Sequences.....	4
Working with Arrays and Matrices	5
Add or Delete Elements in Vectors and Matrices	8
Reading Data Sets in R	9
Configuring the R Workspace	9
Working with data sets that are available in R.....	10
Read TXT files with read.table()	11
Read CSV Excel Files into R	11
Built-in Functions	13
Some Useful Built-in Functions for Vectors:.....	13
Some Useful Built-in Functions for Matrices:.....	14
Data Frame	15
Creating A Data Frame from Vectors	15
Changing class of the object	16
Accessing data from Data Frame.....	17
Data Subsetting	17
Control Structures	21
if and else.....	22
for loops	23
Nested for loops.....	24
while	25
repeat loops and break	25
loops and <i>next</i> commands.....	25
Functions.....	26
return in functions	27
Named Parameters and Default Parameters	28
The plot() function	30
Working with RMarkdown.....	33

Introduction

This document highlights some of the basic and core skills that may be useful for the MSC BDA students; examples are provided where possible.

The skills that this document details are not meant to be a comprehensive set of skills nor a mandatory set of skills. Rather, they indicate a reasonable starting point.

Mathematical Operators and Functions

Examples include:

```
5 + 5 # Addition
## [1] 10

5*5 # Multiplication
## [1] 25

55/5 # Division
## [1] 11

5^2 # Square
## [1] 25

# Functions:
log(5)#Log
## [1] 1.609438

log10(1000)#Log base 10
## [1] 3

exp(5) # exponential
## [1] 148.4132
```

Working with variables:

```
5^2 #taking the square
## [1] 25

a = 3
b = a^2
print(b)#used to print your variable
## [1] 9
```

Logical Operators

Usage examples:

- 1) to check whether a condition is True (T) or False (F).
- 2) to subset a data use specified criteria
- 3) to control the flow of a program e.g. in loops, functions, etc.

Symbols:

> for 'greater than', < for 'less than', >= for 'greater than or equals', <= for 'less than or equals', == for 'equality', ~= for 'inequality', | for 'Or', & for 'And'

Examples:

```
3>6 # 3 greater than 5?
## [1] FALSE

# '&' to check whether both conditions are met
3<6&8>9
## [1] FALSE

# '|' whether either condition is TRUE?
3<6|8>9
## [1] TRUE
```

Working with Sequences

A sequence is a collection of objects (e.g. numbers) in which repetitions are allowed.

The 'seq' function or ':' is used to create sequences.

Examples:

```
a=1:12
a
## [1] 1 2 3 4 5 6 7 8 9 10 11 12
a=seq(5,12)

b=seq(1,12,2) #the last number in a parameter that specifies the increment, here steps of 2.

# To repeat the sequence use the command 'rep'
rep(b,2)
```

```
## [1] 1 3 5 7 9 11 1 3 5 7 9 11
rep(seq(1,7,2),3) # the sequence is repeated 3 times
## [1] 1 3 5 7 1 3 5 7 1 3 5 7
rep(seq(1,7,2),each=3) # each number is repeated 3 times
## [1] 1 1 1 3 3 3 5 5 5 7 7 7
```

Working with Arrays and Matrices

An array object contains a collection of elements of the same type. The elements are indexed (i.e. have ID numbers).

A matrix is a collection of elements of the same type, organized in the form of a table. Each element is indexed by a pair of numbers that identify the row and the column of the element.

Arrays and Matrices can be created using the 'c' (combine) function.

Examples:

```
v=c(1.2,3.5,.79,25)
v
## [1] 1.20 3.50 0.79 25.00
# To extract the rth entry of the vector use v[r] where r is any number
v[3]
## [1] 0.79
v[c(1,2)] # to get more than one entry from vector
## [1] 1.2 3.5
#length function shows the length of an object.
length(v)
## [1] 4
# The logical operators are used to extract elements from array based on some
criteria. e.g
v=c(1.3,3.3,.77,10,25)
v[v>2]
## [1] 3.3 10.0 25.0
v[v>20 & v<30]
## [1] 25
v[v==1.2 | v<1]
```

```
## [1] 0.77

#By using c (combine function), you can create an array that contains only characters.
daga=c("dada","yaya","2020","bibi")
daga

## [1] "dada" "yaya" "2020" "bibi"

# To extract entry from such an array
daga[daga=="yaya"]

## [1] "yaya"

# You can give names to elements in R by using names command.
names(v)=c("entry1","entry2","entry3","entry4", "entry5")
v

## entry1 entry2 entry3 entry4 entry5
## 1.30 3.30 0.77 10.00 25.00

#To construct a matrix.
matx=matrix(c(1,2,3,4,5,6),ncol=2) # arrange in two columns
matx

##      [,1] [,2]
## [1,] 1    4
## [2,] 2    5
## [3,] 3    6

#Or
matx2=matrix(c(1,2,3,4,5,6),3,2)#specify (X by Y) arrangement i.e. 3 rows by 2 columns in this example
matx2

##      [,1] [,2]
## [1,] 1    4
## [2,] 2    5
## [3,] 3    6

#To get an entry by its position (e.g 2,1) in the matrix
matx[2,1]

## [1] 2

# Names can assigned to the rows and columns of the matrix:
rownames(matx)=c("Jan","Feb","Mar")
colnames(matx)=c("day","night")
matx

##      day night
## Jan   1     4
## Feb   2     5
## Mar   3     6
```

```
matx["Jan","day"]
## [1] 1

#You can find out the dimension of any matrix with dim command.
matx

##      day night
## Jan   1     4
## Feb   2     5
## Mar   3     6

dim(matx)#dimension of the matrix
## [1] 3 2

nrow(matx)# Use the nrow command to check the number of rows
## [1] 3

ncol(matx)# Use the ncol command to check the number of columns
## [1] 2
```

Matrix Calculations can be done as follows:

```
# create matrices
A <- matrix(c( 6, 1,+ 0, -3,-1, 2),3, 2, byrow = TRUE)

B <- matrix(c( 4, 2,0, 1,-5, -1),3, 2, byrow = TRUE)

A + B #summation of two matrices

##      [,1] [,2]
## [1,]   10    3
## [2,]    0   -2
## [3,]   -6    1

A - B #subtraction of two matrices

##      [,1] [,2]
## [1,]    2   -1
## [2,]    0   -4
## [3,]    4    3

A * B # this is component-by-component multiplication, not matrix multiplication

##      [,1] [,2]
## [1,]   24    2
## [2,]    0   -3
## [3,]    5   -2

t(A) #take the transpose of the matrix
```

```
##      [,1] [,2] [,3]
## [1,]    6    0   -1
## [2,]    1   -3    2

C<-matrix(c(2,4,5,6),nrow=2)
C

##      [,1] [,2]
## [1,]    2    5
## [2,]    4    6

A%%C #the matrix multiplication

##      [,1] [,2]
## [1,]   16   36
## [2,]  -12  -18
## [3,]    6    7

solve(C) #take the inverse of the matrix

##      [,1] [,2]
## [1,] -0.75  0.625
## [2,]  0.50 -0.250
```

Add or Delete Elements in Vectors and Matrices

Vectors and matrices have set length and dimensions but they can be changed

```
x <- c(12,5,13,16,8)

x <- c(x,20) # append 20 to x

x <- c(x[1:3],20,x[4:6]) # insert 20

x <- x[-2:-4] # delete elements 2 through 4

#The rbind() and cbind() functions enable one to add rows or columns to a matrix.
one=c(1,1,1,1)

z=matrix(c(1,2,3,4,1,1,0,0,1,0,1,0),ncol=3)

cbind(one,z) #"combine vector and matrix as column"

##      one
## [1,]  1 1 1 1
## [2,]  1 2 1 0
## [3,]  1 3 0 1
## [4,]  1 4 0 0

q=rbind(c(1,2),c(3,4)) # combine vector and matrix as row
```

```

# To delete an entry from matrices or vectors - (minus) sign is used.
z=matrix(c(1,2,3,4,1,1,0,0,1,0,1,0),ncol=3)

z[-1,] # deleting first row

##      [,1] [,2] [,3]
## [1,]    2    1    0
## [2,]    3    0    1
## [3,]    4    0    0

z[-c(1,2),]# deleting first and second row. To delete more than one row,

##      [,1] [,2] [,3]
## [1,]    3    0    1
## [2,]    4    0    0

z[, -1]# to delete the first column

##      [,1] [,2]
## [1,]    1    1
## [2,]    1    0
## [3,]    0    1
## [4,]    0    0

z[, -c(1,2)]# deleting column 1 and 2.note the 'c' command is used

## [1] 1 0 1 0

```

Reading Data Sets in R

Important things to note when handling files:

- With spreadsheets, the first row is usually reserved for the header, while the first column is used as ID of row;
- Avoid blank spaces when naming fields and values (spaces sometimes indicate separation) to avoid errors;
- Short names are preferable;
- avoid names that contain symbols such as ?, \$, %, ^, &, *, (,), -, #, ?, <, >, /, |, , [,] , {, and };

Configuring the R Workspace

For convenience put files that you will be read in the working directory.

To check your current working directory use :

```
getwd()
```

To set a new Directory, use one of the following:

- (1) In R Studio: **'Session'** Tab – **'Set Working Directory'**
- (2) R Console: using **setwd()** command, e.g. `setwd("c:/Documents/my/myworkdirectory")`

Working with data sets that are available in R

R and R Studio includes many data sets that are built-in.

To check the current list:

```
ls(name="package:datssets")
```

## [1] "ability.cov"	"airmiles"	"AirPassengers"
## [4] "airquality"	"anscombe"	"attenu"
## [7] "attitude"	"austres"	"beaver1"
## [10] "beaver2"	"BJsales"	"BJsales.lead"
## [13] "BOD"	"cars"	"ChickWeight"
## [16] "chickwts"	"co2"	"CO2"
## [19] "crimtab"	"discoveries"	"DNase"
## [22] "esoph"	"euro"	"euro.cross"
## [25] "eurodist"	"EuStockMarkets"	"faithful"
## [28] "fdeaths"	"Formaldehyde"	"freeny"
## [31] "freeny.x"	"freeny.y"	"HairEyeColor"
## [34] "Harman23.cor"	"Harman74.cor"	"Indometh"
## [37] "infert"	"InsectSprays"	"iris"
## [40] "iris3"	"islands"	"JohnsonJohnson"
## [43] "LakeHuron"	"ldeaths"	"lh"
## [46] "LifeCycleSavings"	"Loblolly"	"longley"
## [49] "lynx"	"mdeaths"	"morley"
## [52] "mtcars"	"nhtemp"	"Nile"
## [55] "nottem"	"npk"	"occupationalStatus"
## [58] "Orange"	"OrchardSprays"	"PlantGrowth"
## [61] "precip"	"presidents"	"pressure"
## [64] "Puromycin"	"quakes"	"randu"
## [67] "rivers"	"rock"	"Seatbelts"
## [70] "sleep"	"stack.loss"	"stack.x"
## [73] "stackloss"	"state.abb"	"state.area"
## [76] "state.center"	"state.division"	"state.name"
## [79] "state.region"	"state.x77"	"sunspot.month"
## [82] "sunspot.year"	"sunspots"	"swiss"
## [85] "Theoph"	"Titanic"	"ToothGrowth"
## [88] "treering"	"trees"	"UCBAdmissions"
## [91] "UKDriverDeaths"	"UKgas"	"USAccDeaths"
## [94] "USArrests"	"UScitiesD"	"USJudgeRatings"
## [97] "USPersonalExpenditure"	"uspop"	"VADeaths"
## [100] "volcano"	"warpbreaks"	"women"
## [103] "WorldPhones"	"WWWusage"	

To Load Built-in data

Type a name to load the data set:

```
# Load the AirPassengers data set.
AirPassengers
```

```
##      Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
## 1949 112 118 132 129 121 135 148 148 136 119 104 118
## 1950 115 126 141 135 125 149 170 170 158 133 114 140
## 1951 145 150 178 163 172 178 199 199 184 162 146 166
## 1952 171 180 193 181 183 218 230 242 209 191 172 194
## 1953 196 196 236 235 229 243 264 272 237 211 180 201
## 1954 204 188 235 227 234 264 302 293 259 229 203 229
## 1955 242 233 267 269 270 315 364 347 312 274 237 278
## 1956 284 277 317 313 318 374 413 405 355 306 271 306
## 1957 315 301 356 348 355 422 465 467 404 347 305 336
## 1958 340 318 362 348 363 435 491 505 404 359 310 337
## 1959 360 342 406 396 420 472 548 559 463 407 362 405
## 1960 417 391 419 461 472 535 622 606 508 461 390 432
```

Read TXT files with read.table()

If you have a .txt or a tab-delimited text file, you can easily import it with the basic R function read.table().

```
# If the columns in the data set have names/headers.
read.table("data.txt",header=T)
```

```
##      ht  wt
## 1  58 115
## 2  59 117
## 3  60 120
## 4  61 123
## 5  62 126
## 6  63 129
## 7  64 132
## 8  65 135
## 9  66 139
## 10 67 142
```

```
# If the columns do not have names.
read.table("data1.txt",header=F)
```

```
##      V1  V2
## 1  58 115
## 2  59 117
## 3  60 120
## 4  61 123
## 5  62 126
## 6  63 129
## 7  64 132
## 8  65 135
## 9  66 139
## 10 67 142
```

Read CSV Excel Files into R

File with values separated by a ',' or ';' are usually '.csv' files.

To load the file into R, you can use the the functions `read.table()`, `read.csv()` or `read.csv2()` functions.

```
autodata=read.table("autoPrice.csv",sep=";",header=T)
head(autodata) #shows some first observation of the data set, you can use it
any time
```

	X	symboling	normalized.losses	wheel.base	length	width	height	curb.weight
## 1	0	2	164	99.8	176.6	66.2	54.3	2337
## 2	1	2	164	99.4	176.6	66.4	54.3	2824
## 3	2	1	158	105.8	192.7	71.4	55.7	2844
## 4	3	1	158	105.8	192.7	71.4	55.9	3086
## 5	4	2	192	101.2	176.8	64.8	54.3	2395
## 6	5	0	192	101.2	176.8	64.8	54.3	2395

	engine.size	bore	stroke	compression.ratio	horsepower	peak.rpm	city.mpg
## 1	109	3.19	3.4	10.0	102	5500	24
## 2	136	3.19	3.4	8.0	115	5500	18
## 3	136	3.19	3.4	8.5	110	5500	19
## 4	131	3.13	3.4	8.3	140	5500	17
## 5	108	3.50	2.8	8.8	101	5800	23
## 6	108	3.50	2.8	8.8	101	5800	23

	highway.mpg	target
## 1	30	13950
## 2	22	17450
## 3	25	17710
## 4	20	23875
## 5	29	16430
## 6	29	16925


```
diadata=read.csv("diabetes1.csv",header=F)
head(diadata)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9
## 1	6	148	72	35	0	33.6	0.627	50	1
## 2	1	85	66	29	0	26.6	0.351	31	0
## 3	8	183	64	0	0	23.3	0.672	32	1
## 4	1	89	66	23	94	28.1	0.167	21	0
## 5	0	137	40	35	168	43.1	2.288	33	1
## 6	5	116	74	0	0	25.6	0.201	30	0

For Delimited Files (data is organized in a data matrix) you can use `read.delim()`. The delimiter can be specified:

`sep="\t"` for tab-delimited

`sep=" "` for space-delimited

`sep=","` for comma-delimited.

```
d=read.delim("data2.txt", header=TRUE, sep="\t")
head(d)
```

	drug	math	ed
## 1	2.17	7.9`	2

```
## 2 2.97 5.20 1
## 3 3.26 6.47 2
## 4 2.69 3.07 3
## 5 3.83 4.15 4
## 6 2.00 2.02 2
```

There other ways of loading data set.

Built-in Functions

To make development easy, many useful functions are built-in.

Some Useful Built-in Functions for Vectors:

```
# define a vector x
x=c(1,3,9,5,9,0,5,6)

length(x) # to get the Length of the vector x
## [1] 8

max(x) # to get the element with maximum value in the vector
## [1] 9

min(x) # to get the minimum value
## [1] 0

which(x==3)# to get the Location of 3
## [1] 2

which.max(x)# to get the Location of the element with the maximum value
## [1] 3

range(x)# get the minimum and maximum values
## [1] 0 9

sum(x) # get the sum up all elements
## [1] 38

cumsum(x)# get cumulative sum of vector
## [1] 1 4 13 18 27 27 32 38

mean(x) # get mean of the vector
## [1] 4.75

median(x)# get median of the vector
## [1] 5
```

```

var(x)# variance of vector
## [1] 11.07143

sd(x)# standard deviation of the vector
## [1] 3.327376

sort(x)# sort in increasing order
## [1] 0 1 3 5 5 6 9 9

sort(x,decreasing = T)# sort in decreasing order
## [1] 9 9 6 5 5 3 1 0

diff(x)# take the difference of ith and (i+1)th element
## [1] 2 6 -4 4 -9 5 1

```

Some Useful Built-in Functions for Matrices:

```

# Define a matrix xy
xy=matrix(c(2,4,2,3,6,1,3,6,9),3,3)

xy

##      [,1] [,2] [,3]
## [1,]    2    3    3
## [2,]    4    6    6
## [3,]    2    1    9

which(xy==6)#get the location of 6 in the matrix as vector xy
## [1] 5 8

which.max(xy)#get the location of the maximum element of the matrix as vector
xy
## [1] 9

length(xy) #get the Length of the matrix as vector
## [1] 9

max(xy) # get the maximum value of the matrix as vector
## [1] 9

min(xy) #get the minimum value of the matrix as vector
## [1] 1

range(xy)#get minimum and maximum values of the matrix as vector
## [1] 1 9

sum(xy)#get the summation of all elements of the matrix as vector

```

```
## [1] 36
cumsum(xy)# get cumulative sum of the matrix as vector
## [1]  2  6  8 11 17 18 21 27 36
mean(xy)#get mean of the matrix as vector
## [1] 4
median(xy)#get median of the matrix as vector
## [1] 3
sd(xy)# get the standard deviation of the matrix as vector
## [1] 2.54951
sort(xy)#sort in ascending the matrix as vector
## [1] 1 2 2 3 3 4 6 6 9
sort(xy,decreasing = T)#sort in descending order
## [1] 9 6 6 4 3 3 2 2 1
diff(xy)#take the difference of i,j th and (i+1,j)th element of the matrix. i
and j are position coordinates.
##      [,1] [,2] [,3]
## [1,]    2    3    3
## [2,]   -2   -5    3
```

Data Frame

A data frame in R combines features of vectors, matrices, and lists. Like vectors, data frames must have the same kind of data in each column. Like matrices, data frames have both rows and columns. Like lists, data frames allow the user to have a combination of numeric, character, and logical data.

A data frame may be likened to a worksheet in Excel (or some other spreadsheet program) or a statistics program like SPSS or JMP.

Creating A Data Frame from Vectors

```
people <-c("student1","student2","student3","student4","student5","student6",
"student7","student8","student9","student10")

gender<-c("m","m","m","f","f","m","f","f","f","m")

scores <-c(17,19,16,15,23,17,24,29,24,25)

quiz_scores <- data.frame(people,gender,scores) #to create a data frame
```

```
quiz_scores

##      people gender scores
## 1  student1      m     17
## 2  student2      m     19
## 3  student3      m     16
## 4  student4      f     15
## 5  student5      f     23
## 6  student6      m     17
## 7  student7      f     24
## 8  student8      f     29
## 9  student9      f     24
## 10 student10     m     25
```

We can obtain individual columns by using the column index in square brackets. We can also employ the data frame name followed by a \$ sign and the column name.

```
quiz_scores[2] # get column 2 i.e. gender

##      gender
## 1         m
## 2         m
## 3         m
## 4         f
## 5         f
## 6         m
## 7         f
## 8         f
## 9         f
## 10        m

quiz_scores$scores # get the scores column

## [1] 17 19 16 15 23 17 24 29 24 25
```

Changing class of the object

Example - change the class of an object after reading it.

```
x=1:5 # create a sequence

y=6:10 # create a sequence

xy=c(x,y) # combine the sequences with 'c' command

class(xy)#check the class of the object

## [1] "integer"

xy=as.data.frame(xy) # change the class of 'xy' to a data frame
class(xy)

## [1] "data.frame"
```

```
xy1=as.matrix(xy) # create a matrix with xy
class(xy1)
```

```
## [1] "matrix" "array"
```

Accessing data from Data Frame

```
quiz_scores$people
```

```
## [1] "student1" "student2" "student3" "student4" "student5" "student6"
```

```
## [7] "student7" "student8" "student9" "student10"
```

```
quiz_scores$scores>15
```

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

```
quiz_scores$scores[quiz_scores$scores>15] #to see grades greater than 15
```

```
## [1] 17 19 16 23 17 24 29 24 25
```

```
quiz_scores[quiz_scores[, "scores"]>15,] #to see grades greater than 15 and co  
rresponding other variables
```

```
##      people gender scores
## 1 student1      m      17
## 2 student2      m      19
## 3 student3      m      16
## 5 student5      f      23
## 6 student6      m      17
## 7 student7      f      24
## 8 student8      f      29
## 9 student9      f      24
## 10 student10     m      25
```

```
quiz_scores$scores[quiz_scores$gender=="f"]#to see grades for female students
```

```
## [1] 15 23 24 29 24
```

```
quiz_scores[quiz_scores[, "gender"]=="f",]#to see grades for female students a  
nd corresponding other variables
```

```
##      people gender scores
## 4 student4      f      15
## 5 student5      f      23
## 7 student7      f      24
## 8 student8      f      29
## 9 student9      f      24
```

Data Subsetting

Data subsetting is an important part of the data analysis.

```
data=read.table("data3.txt",header=T) # Load data with demographic informatio  
n
```

```
head(data)
```

```
##      Qtr1  Qtr2 Qtr3  Qtr4
## 1960 160.1 129.7 84.8 120.1
## 1961 160.1 124.9 84.8 116.9
## 1962 169.7 140.9 89.7 123.3
## 1963 187.3 144.1 92.9 120.1
## 1964 176.1 147.3 89.7 123.3
## 1965 185.7 155.3 99.3 131.3
```

```
class(data) #check the class of object
```

```
## [1] "data.frame"
```

```
dim(data) # check the dimensions
```

```
## [1] 27  4
```

```
# SUBSET BY ROWS
```

```
# use a sequence to specify a subset
```

```
data[1:5,]
```

```
##      Qtr1  Qtr2 Qtr3  Qtr4
## 1960 160.1 129.7 84.8 120.1
## 1961 160.1 124.9 84.8 116.9
## 1962 169.7 140.9 89.7 123.3
## 1963 187.3 144.1 92.9 120.1
## 1964 176.1 147.3 89.7 123.3
```

```
# use a 'c' command to specify a subset
```

```
data[c(1,3,5,7),]
```

```
##      Qtr1  Qtr2  Qtr3  Qtr4
## 1960 160.1 129.7  84.8 120.1
## 1962 169.7 140.9  89.7 123.3
## 1964 176.1 147.3  89.7 123.3
## 1966 200.1 161.7 102.5 136.1
```

```
# specify what to exclude
```

```
data[-4,] # exclude 4th row
```

```
##      Qtr1  Qtr2  Qtr3  Qtr4
## 1960 160.1 129.7  84.8 120.1
## 1961 160.1 124.9  84.8 116.9
## 1962 169.7 140.9  89.7 123.3
## 1964 176.1 147.3  89.7 123.3
## 1965 185.7 155.3  99.3 131.3
## 1966 200.1 161.7 102.5 136.1
## 1967 204.9 176.1 112.1 140.9
## 1968 227.3 195.3 115.3 142.5
## 1969 244.9 214.5 118.5 153.7
## 1970 244.9 216.1 188.9 142.5
## 1971 301.0 196.9 136.1 267.3
```

```
## 1972 317.0 230.5 152.1 336.2
## 1973 371.4 240.1 158.5 355.4
## 1974 449.9 286.6 179.3 403.4
## 1975 491.5 321.8 177.7 409.8
## 1976 593.9 329.8 176.1 483.5
## 1977 584.3 395.4 187.3 485.1
## 1978 669.2 421.0 216.1 509.1
## 1979 827.7 467.5 209.7 542.7
## 1980 840.5 414.6 217.7 670.8
## 1981 848.5 437.0 209.7 701.2
## 1982 925.3 443.4 214.5 683.6
## 1983 917.3 515.5 224.1 694.8
## 1984 989.4 477.1 233.7 730.0
## 1985 1087.0 534.7 281.8 787.6
## 1986 1163.9 613.1 347.4 782.8
```

```
data[-c(1,3,5,7),]# exclude rows 1,3,5,7
```

```
##      Qtr1  Qtr2  Qtr3  Qtr4
## 1961 160.1 124.9  84.8 116.9
## 1963 187.3 144.1  92.9 120.1
## 1965 185.7 155.3  99.3 131.3
## 1967 204.9 176.1 112.1 140.9
## 1968 227.3 195.3 115.3 142.5
## 1969 244.9 214.5 118.5 153.7
## 1970 244.9 216.1 188.9 142.5
## 1971 301.0 196.9 136.1 267.3
## 1972 317.0 230.5 152.1 336.2
## 1973 371.4 240.1 158.5 355.4
## 1974 449.9 286.6 179.3 403.4
## 1975 491.5 321.8 177.7 409.8
## 1976 593.9 329.8 176.1 483.5
## 1977 584.3 395.4 187.3 485.1
## 1978 669.2 421.0 216.1 509.1
## 1979 827.7 467.5 209.7 542.7
## 1980 840.5 414.6 217.7 670.8
## 1981 848.5 437.0 209.7 701.2
## 1982 925.3 443.4 214.5 683.6
## 1983 917.3 515.5 224.1 694.8
## 1984 989.4 477.1 233.7 730.0
## 1985 1087.0 534.7 281.8 787.6
## 1986 1163.9 613.1 347.4 782.8
```

```
# SUBSET BY COLUMNS
```

```
# Specify the column to exclude
```

```
data[, -2] # exclude column 2
```

```
##      Qtr1  Qtr3  Qtr4
## 1960 160.1  84.8 120.1
## 1961 160.1  84.8 116.9
## 1962 169.7  89.7 123.3
## 1963 187.3  92.9 120.1
## 1964 176.1  89.7 123.3
```

```
## 1965 185.7 99.3 131.3
## 1966 200.1 102.5 136.1
## 1967 204.9 112.1 140.9
## 1968 227.3 115.3 142.5
## 1969 244.9 118.5 153.7
## 1970 244.9 188.9 142.5
## 1971 301.0 136.1 267.3
## 1972 317.0 152.1 336.2
## 1973 371.4 158.5 355.4
## 1974 449.9 179.3 403.4
## 1975 491.5 177.7 409.8
## 1976 593.9 176.1 483.5
## 1977 584.3 187.3 485.1
## 1978 669.2 216.1 509.1
## 1979 827.7 209.7 542.7
## 1980 840.5 217.7 670.8
## 1981 848.5 209.7 701.2
## 1982 925.3 214.5 683.6
## 1983 917.3 224.1 694.8
## 1984 989.4 233.7 730.0
## 1985 1087.0 281.8 787.6
## 1986 1163.9 347.4 782.8
```

```
data[,-c(2,3)] # exclude a set of columns using the 'c' combine command
```

```
##      Qtr1 Qtr4
## 1960 160.1 120.1
## 1961 160.1 116.9
## 1962 169.7 123.3
## 1963 187.3 120.1
## 1964 176.1 123.3
## 1965 185.7 131.3
## 1966 200.1 136.1
## 1967 204.9 140.9
## 1968 227.3 142.5
## 1969 244.9 153.7
## 1970 244.9 142.5
## 1971 301.0 267.3
## 1972 317.0 336.2
## 1973 371.4 355.4
## 1974 449.9 403.4
## 1975 491.5 409.8
## 1976 593.9 483.5
## 1977 584.3 485.1
## 1978 669.2 509.1
## 1979 827.7 542.7
## 1980 840.5 670.8
## 1981 848.5 701.2
## 1982 925.3 683.6
## 1983 917.3 694.8
## 1984 989.4 730.0
```

```
## 1985 1087.0 787.6
## 1986 1163.9 782.8

# SUBSET WITH LOGICAL OPERATORS
data[data$Qtr1 >=500,] # specify: where Qtr1 values are greater than 500

##      Qtr1  Qtr2  Qtr3  Qtr4
## 1976  593.9 329.8 176.1 483.5
## 1977  584.3 395.4 187.3 485.1
## 1978  669.2 421.0 216.1 509.1
## 1979  827.7 467.5 209.7 542.7
## 1980  840.5 414.6 217.7 670.8
## 1981  848.5 437.0 209.7 701.2
## 1982  925.3 443.4 214.5 683.6
## 1983  917.3 515.5 224.1 694.8
## 1984  989.4 477.1 233.7 730.0
## 1985 1087.0 534.7 281.8 787.6
## 1986 1163.9 613.1 347.4 782.8

data[data[,1]>=500,]# you can reference the column by its number Qtr1 is column 1

##      Qtr1  Qtr2  Qtr3  Qtr4
## 1976  593.9 329.8 176.1 483.5
## 1977  584.3 395.4 187.3 485.1
## 1978  669.2 421.0 216.1 509.1
## 1979  827.7 467.5 209.7 542.7
## 1980  840.5 414.6 217.7 670.8
## 1981  848.5 437.0 209.7 701.2
## 1982  925.3 443.4 214.5 683.6
## 1983  917.3 515.5 224.1 694.8
## 1984  989.4 477.1 233.7 730.0
## 1985 1087.0 534.7 281.8 787.6
## 1986 1163.9 613.1 347.4 782.8

# select rows that fall within a range of values in Qtr2 column
data[data$Qtr2>=500&data$Qtr2<=800,]

##      Qtr1  Qtr2  Qtr3  Qtr4
## 1983  917.3 515.5 224.1 694.8
## 1985 1087.0 534.7 281.8 787.6
## 1986 1163.9 613.1 347.4 782.8
```

Control Structures

Controlling the flow of your program based on some 'logic' is important in programming, examples include:

- **if** and **else**: can be used to control the logic flow and act based on meeting a condition
- **for**: execute a loop for a fixed number of times
- **while**: execute a loop while a condition is true

- **repeat:** execute an infinite loop until it a *break* or *stop* command is issued
- **break:** break the execution of a loop
- **next:** skip an iteration in a loop

if and else

In this structure, you can test a condition and execute an action based on whether it's true or false.

The **if** and **else** commands can be combined in different ways:

(a) Take action based if a single condition is met

Example:

```
# This example simply check to see if a number is a positive number

a = 0.9
if (a>0){print('found a positive number')}# prints because a is greater than 0

## [1] "found a positive number"

b = -0.9
if (b>0){print('found a positive number')}# does nothing because b is not greater than 0
```

(b) Take action if a condition is met or take another specified action if the condition is met

```
if (condition){ #do this action if condition is true
```

```
    }
```

```
else{ #do this action
```

```
}
```

```
# returns 'yes' when positive and 'no' otherwise
x<- c(0.8, -0.9, -0.6, 0.9, -0.9, 0.8)

y2 <- ifelse(x>0, 'yes', 'no')
y2

## [1] "yes" "no"  "no"  "yes" "no"  "yes"
```

(c) Take an action if a condition is met and a different action if another condition is met. Then, if non of the conditions is met take a specified action is taken.

```
if (condition){
```

```
  #do something if condition is true
```

```
} else if (condition2) {
```

```
  #do something if condition2 is true
```

```
} else {  
#do something if neither condition 1 nor condition 2 is true  
}
```

for loops

Loops can be used to repeat actions, such as iterating over the elements of an object.

```
# Loop through a sequence  
for (i in 10:20) {  
  print(i)  
}  
  
## [1] 10  
## [1] 11  
## [1] 12  
## [1] 13  
## [1] 14  
## [1] 15  
## [1] 16  
## [1] 17  
## [1] 18  
## [1] 19  
## [1] 20  
  
x <- c("item1", "item2", "item3", "item4", "item5")  
  
# Loop through a list of items  
for (i in x) {  
  print(i)  
}  
  
## [1] "item1"  
## [1] "item2"  
## [1] "item3"  
## [1] "item4"  
## [1] "item5"  
  
# use the length of the vector to determine how many times to loop  
for (i in 1:length(x)) {  
  print(i)  
}  
  
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5
```

An example that applies both *loops* and *if else* structures.

```

v=c(5,-6,2,0,-2,4)

for (i in 1:length(v))
if(v[i]>0){
print("'found a positive number'")

} else if (v[i]<0){
print("'found a negative number")
} else{
print("equals to zero")
}

## [1] "'found a positive number'"
## [1] "'found a negative number"
## [1] "'found a positive number'"
## [1] "equals to zero"
## [1] "'found a negative number"
## [1] "'found a positive number'"

```

Nested for loops

for loops can be nested inside of each other.

```

# create a matrix
m <- matrix(1:12, 3)
m

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

# Loop by row and then column using the 'nrow' and 'ncol' functions
for (i in 1:nrow(m)) {
  for (j in 1:ncol(m)) {
    print(m[i, j]) # print the items
  }
}

## [1] 1
## [1] 4
## [1] 7
## [1] 10
## [1] 2
## [1] 5
## [1] 8
## [1] 11
## [1] 3
## [1] 6
## [1] 9
## [1] 12

```

Note: avoid using too many nested loops by using functions.

while

Loops continuously until a condition is met. This means it checks on each loop to see If the condition is true.

```
# This example starts from 0 increments a variable and only stops when the incremental variable reaches 10.
count<-0
while(count<10){ # continue while the condition is true
count<-count+1
print(count)
}

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

repeat loops and break

Like the *While* loop, it repeats an action until a condition is met.

The *break* command can be used to exit the loop.

```
x <- 1
repeat {
print(x)
x = x+1
if (x == 6){ # i.e. stops if x get to the value 6
break # the command to exit the loop
}
}

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

loops and next commands

The next command is used to skip an iteration of a loop (either specified or when a criterion is met).

```
# In this example, the first 10 iterations are Skipped
for(i in 1:20) {
if(i <= 10) {
```

```

next
}
  print(i)
}

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

# In this example, the last 9 iterations are Skipped
for(i in 1:20) {
  print(i)
  if(i > 10) {
## Stop Loop after 20 iterations
    break
  }
}

## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11

```

Functions

The **function** keyword is used to create new functions

```

# this function computes the square of numbers
square<-function(x){
  x^2
}

square(2)

## [1] 4

square(-2)

```

```
## [1] 4
```

Functions can be defined to have more than one arguments as input.

```
# In this example x and y are input.
```

```
squares.sum<-function(x,y){  
  (x^2)+(y^2)  
}
```

```
squares.sum(2,3)
```

```
## [1] 13
```

```
# A function to rescale a vector of numbers. Some in-built functions have been used.
```

```
rescale <- function(x) {  
  m <- mean(x) # mean of the input  
  s <- sd(x)   # standard deviation of input  
  (x - m) / s # calculate the scaled values  
}
```

```
x<-c(5,6,3,9,6)
```

```
rescale(x)
```

```
## [1] -0.36901248  0.09225312 -1.29154369  1.47604993  0.09225312
```

Besides the built functions, user defined function and control structures can be used when defining a new function.

```
# Define a function that returns the square of a number, 'square'
```

```
square<-function(x){  
  x^2  
}
```

```
# A new function ('equation') is defined that uses 'square'
```

```
equation<-function(x){  
  square(x)+(3*x)+5  
}
```

```
equation (2)
```

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

return in functions

For functions that produce an output, how do we decide what to return as output?

A common practice is to return a value by just making it the last expression in a function.

Alternatively, you can explicitly return a value from a function before its last expression by using the 'return' function. The 'return' function is usually used to exit a function early.

Examples:

```
x<-c(5,6,3,9,6)

# This example returns the last line
rescale <- function(x) {
  m <- mean(x)
  s <- sd(x)
  (x - m) / s
  (x + m)/s
}

rescale(x)

## [1] 4.981669 5.442934 4.059137 6.826731 5.442934

# This example explicit specifies what to return
rescale <- function(x) {
  m <- mean(x)
  s <- sd(x)
  (x - m) / s
  (x + m)/s
  return((x - m) / s)
}

rescale(x)

## [1] -0.36901248 0.09225312 -1.29154369 1.47604993 0.09225312

# This example explicit specifies what to return
rescale <- function(x) {
  m <- mean(x)
  s <- sd(x)
  scale<-(x - m) / s
  notscale<-(x+m)/s
  return(scale)
}

rescale(x)

## [1] -0.36901248 0.09225312 -1.29154369 1.47604993 0.09225312
```

Named Parameters and Default Parameters

Functions can be defined to accept a parameter. For example, a temperature conversion function can have a parameter that specifies whether the input value is to convert from Celsius to Fahrenheit or vice versa.

In the example below, the function expects (as input) both the value to be converted and the parameter that specifies the type of conversion.

The parameter can be defined to have a default behaviour that it can use when the parameter is not provided.

```

# CREATE THE SUB FUNCTIONS -----
# Returns Fahrenheit conversion of passed c.temp (in Celsius)
convert.to.far <- function(c.temp) {
  f.temp <- c.temp * 9/5 + 32
  return(f.temp)
}

# Returns Celsius conversion of passed f.temp (in Fahrenheit)
convert.to.cel <- function(f.temp)
{
  c.temp <- (f.temp - 32) * 5/9
  return(c.temp)
}

# PUT IT TOGETHER -----
# Define the conversion function
convert.temp <- function(temp, to.celsius = TRUE) { # to.celsius = TRUE speci
fies the default behaviour
  if (to.celsius) {
    converted <- convert.to.cel(temp)
  } else {

converted <- convert.to.far(temp)
  }
  return(converted)
}

# TESTING -----
# Values to be converted
temp<-c(15,26,35,37,26,4,-2)

convert.temp(temp,to.celsius = TRUE)# TRUE/T means convert to Celsius

## [1] -9.444444 -3.333333  1.666667  2.777778 -3.333333 -15.555556 -18.
888889

convert.temp(temp,to.celsius = F)# FALSE/F means convert to Fahrenheit

## [1] 59.0 78.8 95.0 98.6 78.8 39.2 28.4

# When the parameter is not given it uses the default value
convert.temp(temp)# the default value of 'to.celsius' was defined as True i.e
. convert to Celsius

## [1] -9.444444 -3.333333  1.666667  2.777778 -3.333333 -15.555556 -18.
888889

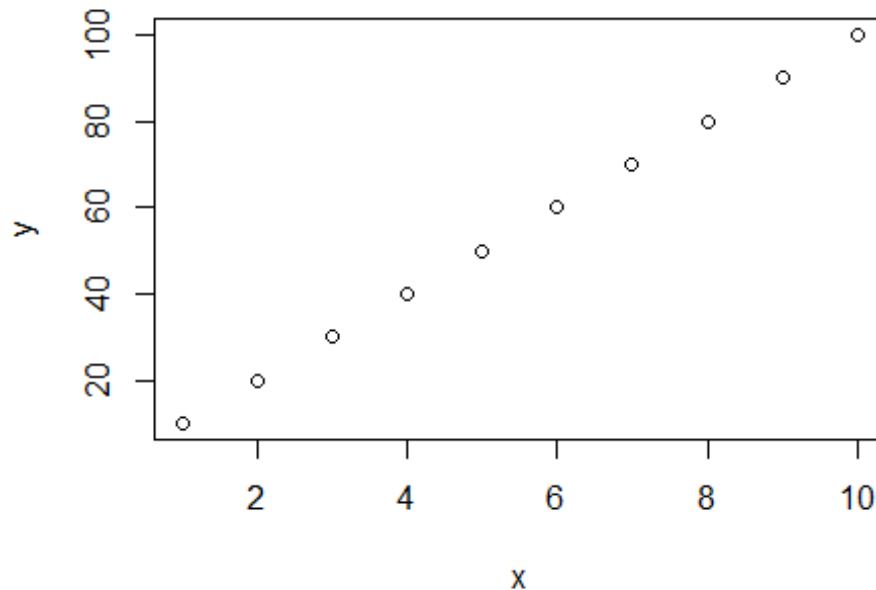
```

The plot() function

R is rich with powerful data visualisation techniques and packages that a data scientist may find useful.

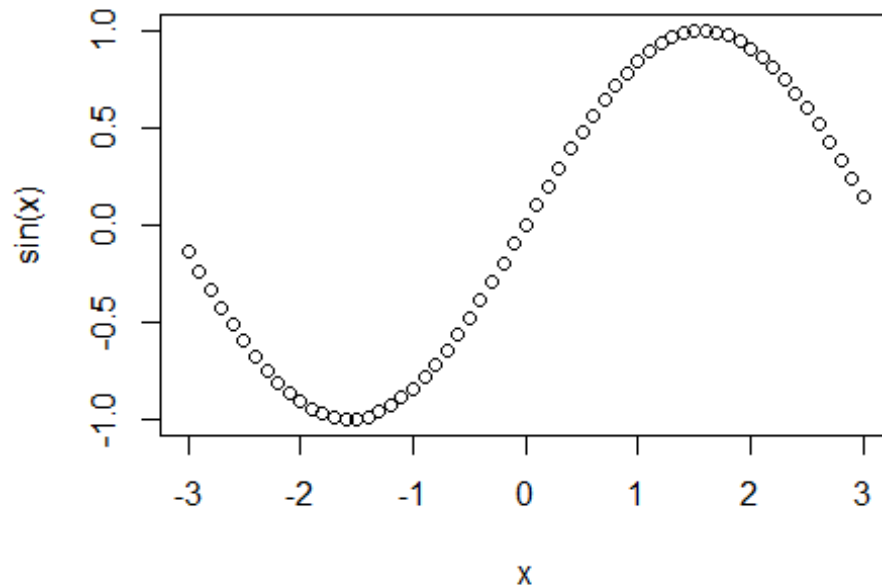
Data visualisation tools and needs of users are very diverse. Therefore, it is not practical to provide an overview in a few pages. However, a basic plotting tool that may be useful is the 'plot()' function; it can work with a variety of objects e.g. vectors and functions.

```
# In examples plot() is receiving vectors that represent the x and y  
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
y <- c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)  
  
plot(x, y)
```



Here is a more concrete example where we plot a sine function from range -pi to pi.

```
# In this example, plot() receives a vector of values as x and calculates the  
y values using the sin() function.  
  
x <- seq(-3,3,0.1) # a sequence between -3 and 3 with intervals of 0.1  
  
plot(x, sin(x)) # y is sin(x)
```



```
# R Plot Function Example
```

Setting Titles and Labeling Axes

Use the *main* parameter to set the title.

Use the *xlab* and *ylab* parameters to set the x-axis and y-axis labels, respectively.

Changing Plot types

Use the *type* parameter to change the plot type.

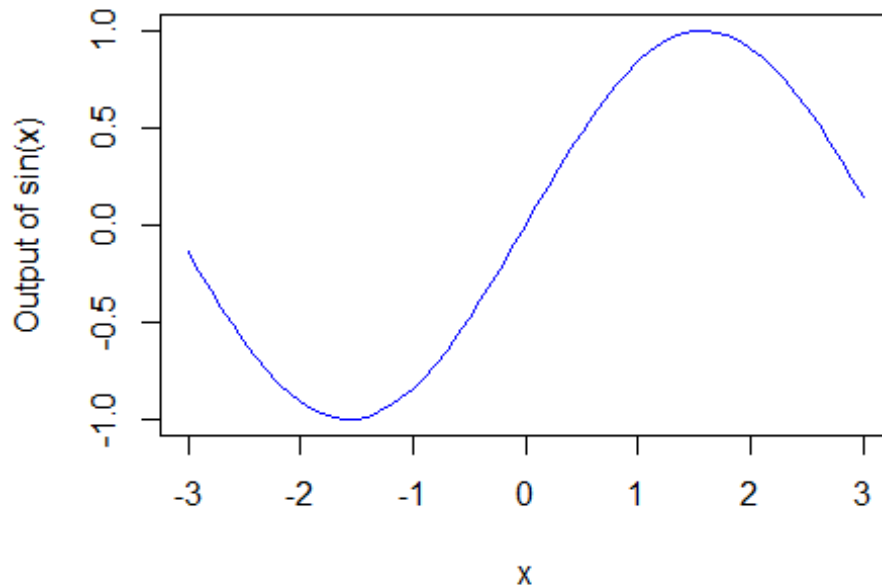
Plot type options include: "p" - points "l" - lines "b" - both points and lines "c" - empty points joined by lines "o" - overplotted points and lines "s" - stair steps "h" - histogram-like vertical lines "n" - does not produce any points or lines

Changing the colour of plots

use parameter **col** to set colour e.g. col="blue".

```
plot(x, sin(x),  
main="The Sine Function",  
ylab="Output of sin(x)", # set y axis title  
type="l", # set plot type to line  
col="blue")
```

The Sine Function



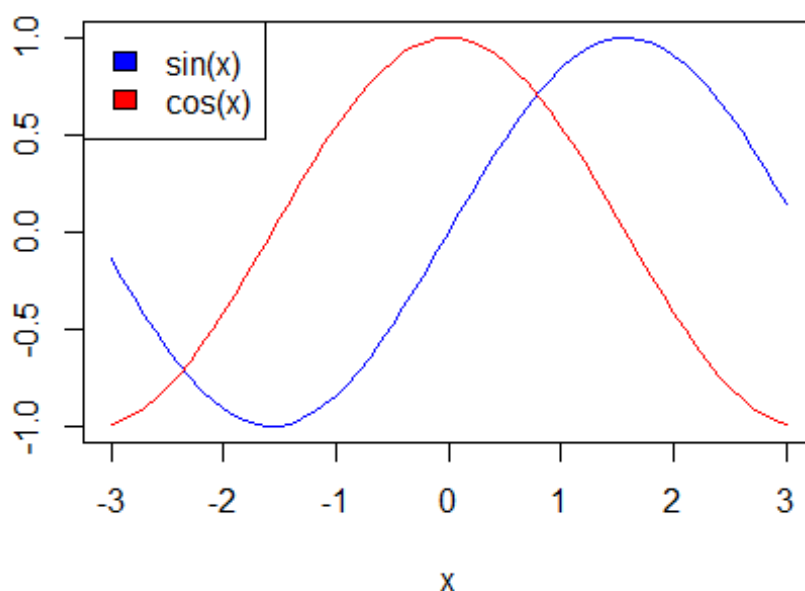
Coloring a plot in R programming

Overlaying Plots and Using the legend() function

Use the lines() and points() functions to add more lines and points to a plot.

```
plot(x, sin(x),  
main="Overlaying Graphs",  
ylab="",  
type="l",  
col="blue")  
  
lines(x,cos(x), col="red") # additional line plot  
  
legend("topleft",# the legend is defined - position, text & colour  
c("sin(x)","cos(x)"),  
fill=c("blue","red")  
)
```

Overlaying Graphs



```
# Overlaying plots in R Using legend() function
```

Getting Help

It is important to learn to use help on commands and packages. The extensive help and documentation of R and R packages can be accessed using commands on the console or the GUI of R studio.

Working with RMarkdown

R Markdown was used to create this document.

R Markdown allows you to incorporate your text and code in one document and output it in formats like PDF, MS Word document and HTML. It can be very useful in creating reports and papers that show code. Features, such as the one that allows you to work on snippets of the code and run them make, R Markdown user friendly. See <http://rmarkdown.rstudio.com>.