Introduction to R Programming

Jasper Otieno

Updated on 18/Sept/2023

Contents

This is an introduction to R programming with end goal of learning the basics of R and finally being able to program in R and proceed to advanced programming.

BASIC R PROGRAMMING

DATA TYPES

R stores data in different types or classes. There are about 4 classes/types of data that R can store. To check the kind of class of data you are working with, use the function `class()` and use the function `typeof()` to check the type of data.

Character

Refers to data in form of letters, sentences or numbers formatted to texts/strings e.g, names of people (James, rose), letter A, B, or colours, green, blue. They must always be wrapped in double quotation marks when referenced.

```
(char1 <- c("ann", "erick", "", "1"))
```

```
## [1] "ann"   "erick" ""      "1"
```

```
class(char1)
```

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

```
typeof(char1)
```

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

Numeric

Are class of data that are stored in number formats. These include whole numbers, decimals or even dates. There are various types of numeric data which include integers, double or just numeric.

```
(numeric1 <- c(1, 2, 3, 4, 5, 6.5)) #double
```

```
## [1] 1.0 2.0 3.0 4.0 5.0 6.5
```

```
class(numeric1)
```

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

```
typeof(numeric1)
```

```
## [1] "double"
```

Integers

Are numeric data in form of whole numbers. They are denoted by letter L next to them.

```
(numeric2 <- c(1L, 2L, 3L, 4L, 5L)) #integer (whole numbers)
```

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

```
class(numeric2)
```

```
## [1] "integer"
```

```
typeof(numeric2)
```

```
## [1] "integer"
```

Factor

Refers to class of data that stores data in categorical format. They values take whole numbers hence are integer type.

Logical

These are data types that take the values TRUE or FALSE. They are special kind of numeric data that indicates if a condition is true or not.

```
(lg1 <- c(TRUE, FALSE, FALSE, TRUE))
```

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

```
class(lg1)
```

```
## [1] "logical"
```

```
typeof(lg1)
```

```
## [1] "logical"
```

Dates and Times

Data can be stored in form of dates or times. They are special kind of numeric data of type double.

```
#System Datetime
(date1 <- Sys.time())
```

```
## [1] "2023-09-20 00:42:58 EAT"
```

```
class(date1)
```

```
## [1] "POSIXct" "POSIXt"
```

```
typeof(date1)
```

```
## [1] "double"
```

```
#System date
(date2 <- Sys.Date())
```

```
## [1] "2023-09-20"
```

```
class(date2)
```

```
## [1] "Date"
```

```
typeof(date2)
```

```
## [1] "double"
```

Data Type Conversion

Data can be converted from one type to another depending on need of analysis or for storage.

```
(numeric1 <- c(1, 2, 3, 4, 5, 6.5))
```

```
## [1] 1.0 2.0 3.0 4.0 5.0 6.5
```

```
(numeric2 <- c(1L, 2L, 3L, 4L, 5L))
```

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

```
(data6 <- as.character(numeric1)) #converting numeric to character
```

```
## [1] "1"   "2"   "3"   "4"   "5"   "6.5"
```

```
class(data6)
```

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

```
typeof(data6)
```

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

```
(data7 <- as.numeric(data6)) #converting character to numeric
```

```
## [1] 1.0 2.0 3.0 4.0 5.0 6.5
```

```
class(data7)
```

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

```
typeof(data7)
```

```
## [1] "double"
```

```
(data8 <- as.numeric(numeric2)) #converting integer to numeric
```

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

```
class(data8)
```

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

```
typeof(data8)
```

```
## [1] "double"
```

DATA STRUCTURES

These are ways of organising data for ease of analysis or storage. There are 4 ways that data can be organised in R. To know the structure of your data, use the function `str()`.

Vectors

Are the smallest or easiest way of organising data. They are as simple as just one character or letter. Vectors are single dimensional and homogeneous (only stores one type of data).

```
#numeric vectors
(vect1 <- c(1, 3, 5, 68, 0)) #specified numbers, integers
```

```
## [1]  1  3  5 68  0
```

```
(vect2 <- 1:10) #vector of sequential numbers 1 to 10. Default constant is 1
```

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

```
(vect4 <- 0:30) #vector of numbers increasing from 0 to 30. Reverse is true i.e., 30:0
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [26] 25 26 27 28 29 30
```

```
(vect6 <- 2.5:7.5)
```

```
## [1] 2.5 3.5 4.5 5.5 6.5 7.5
```

```
class(vect1)
```

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

```
typeof(vect1)
```

```
## [1] "double"
```

```
str(vect1)
```

```
##  num [1:5] 1 3 5 68 0
```

```
#character vectors
(vect7 <- c("Jasper", "Dave", "Green", "A", ""))
```

```
## [1] "Jasper" "Dave"   "Green"  "A"      ""
```

```
str(vect7)
```

```
##  chr [1:5] "Jasper" "Dave" "Green" "A" ""
```

```
#Logical vectors
(vect8 <- c(TRUE, TRUE, FALSE, TRUE))
```

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

```
str(vect8)
```

```
##  logi [1:4] TRUE TRUE FALSE TRUE
```

```
#Factor vector
(gender <- c("F", "M", "F", "M", "M"))
```

```
## [1] "F" "M" "F" "M" "M"
```

```
#convert gender to a factor
(gender1 <- factor(gender))
```

```
## [1] F M F M M
## Levels: F M
```

```
(fcoulor <- factor(c("blue", "green", "red", "pink", "green")))
```

```
## [1] blue  green red   pink  green
## Levels: blue green pink red
```

```
(vect9 <- seq(from = 1,
              to = 25,
              by = 2)) #creates sequence of numbers between 1 and 25 increasing by 2
```

Forming sequence of numbers

```
##  [1]  1  3  5  7  9 11 13 15 17 19 21 23 25
```

```r
(vect10 <- seq(from = 1,
               to = 25,
               length.out = 50)) # creates a vector of length 50 (50 elements) from 1 to 25 equally space
```

```
##  [1]  1.000000  1.489796  1.979592  2.469388  2.959184  3.448980  3.938776
##  [8]  4.428571  4.918367  5.408163  5.897959  6.387755  6.877551  7.367347
## [15]  7.857143  8.346939  8.836735  9.326531  9.816327 10.306122 10.795918
## [22] 11.285714 11.775510 12.265306 12.755102 13.244898 13.734694 14.224490
## [29] 14.714286 15.204082 15.693878 16.183673 16.673469 17.163265 17.653061
## [36] 18.142857 18.632653 19.122449 19.612245 20.102041 20.591837 21.081633
## [43] 21.571429 22.061224 22.551020 23.040816 23.530612 24.020408 24.510204
## [50] 25.000000
```

```r
(vect11 <- 0:30)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [26] 25 26 27 28 29 30
```

```r
(length(vect11)) # function length before a vector object outputs vector length
```

```
## [1] 31
```

```r
vect4 * 2 #each element in the vector is multiplied by 2
```

Calculations using vectors

```
##  [1]  0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
## [26] 50 52 54 56 58 60
```

```r
vect4 - 10 #10 is subtracted from each of the elements
```

```
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6   7   8
## [20]   9  10  11  12  13  14  15  16  17  18  19  20
```

```r
vect4 / 5 #each element of vect5 is divided by 4
```

```
##  [1] 0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6
## [20] 3.8 4.0 4.2 4.4 4.6 4.8 5.0 5.2 5.4 5.6 5.8 6.0
```

```r
vect4^5 #each element is raised to the power of 4
```

```
##  [1]        0        1       32      243     1024     3125     7776    16807
##  [9]    32768    59049   100000   161051   248832   371293   537824   759375
## [17]  1048576  1419857  1889568  2476099  3200000  4084101  5153632  6436343
## [25]  7962624  9765625 11881376 14348907 17210368 20511149 24300000
```

```
sqrt(vect4) #outputs square root of each element of vector4
```

```
##  [1] 0.000000 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [9] 2.828427 3.000000 3.162278 3.316625 3.464102 3.605551 3.741657 3.872983
## [17] 4.000000 4.123106 4.242641 4.358899 4.472136 4.582576 4.690416 4.795832
## [25] 4.898979 5.000000 5.099020 5.196152 5.291503 5.385165 5.477226
```

```
sqrt(vect4[31]) #outputs sqrt of element at position 31 of vect4
```

```
## [1] 5.477226
```

Retrieving vector elements Square brackets [] are used to index the position of elements in any data structure, vectors included. Lowest index of an element is 1 and not 0.

```
(vect4 <- 0:61)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [26] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
## [51] 50 51 52 53 54 55 56 57 58 59 60 61
```

```
length(vect4)
```

```
## [1] 62
```

```
vect4[25] #retrieves element at position 25 in vector4
```

```
## [1] 24
```

```
vect4[26] #retrieves element at position 26 in vector 4
```

```
## [1] 25
```

```
vect4[62] #retrieves element at position 26 in vector 4
```

```
## [1] 61
```

```
(data5 <- c(1, 2, 3, "Secondary", "University"))
```

Converting vectors from one type to another

```
## [1] "1"          "2"          "3"          "Secondary"  "University"
```

```
class(data5) #character vector
```

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

```
(x <-as.numeric(data5)) #converting to numeric
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1  2  3 NA NA
```

```
class(x)
```

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

```
(data6 <- c(1, 2, 3))
```

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

```
class(data6) #numeric vector
```

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

```
(y <- as.character(data6)) #converting to character
```

```
## [1] "1" "2" "3"
```

```
class(y)
```

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

Matrices

These are similar to vectors however they are bi dimensional. The data is organised into rows and columns. Columns are clearly defined. Is combination of multiple vectors. Use the function `matrix()` to assign an object a matrix. Numeric matrices can be used to quickly carry out summaries. By default, matrix arranges the elements by the columns, that is the elements increasing from column to column. This can be changed using the `byrow=TRUE` argument. Matrices are homogeneous.

```
(mtx1 <- matrix(c(1:9), nrow=3, ncol=3))
```

Creating a matrix

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

```
#change order of elements
(mtx2 <- matrix(c(1:9), nrow=3, ncol=3, byrow = TRUE))
```

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

```
mtx1
```

Retreiving elements of a matrix

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

```
mtx1[2,2] #pulls the element at row2 in column2
```

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

```
mtx1[,3] #pulls all elements in column 3
```

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

```
mtx1[3,] #pulls all elements in row3
```

```
## [1] 3 6 9
```

```
mtx1[7] #pulls the element at index 7
```

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

```
mtx2[c(1,3),] #pulls everything in rows 1 and 3
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    7    8    9
```

```
mtx2[c(1,3)] #returns values at index 1 and 7
```

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

```r
mtx2[,c(2,3)] #returns column 2 and 3
```

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

```r
mtx2[,1:3] #returns column 2 to 3
```

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

```r
#summary
summary(mtx1)
```

Calculations using matrices

```
##        V1            V2            V3
##  Min.   :1.0   Min.   :4.0   Min.   :7.0
##  1st Qu.:1.5   1st Qu.:4.5   1st Qu.:7.5
##  Median :2.0   Median :5.0   Median :8.0
##  Mean   :2.0   Mean   :5.0   Mean   :8.0
##  3rd Qu.:2.5   3rd Qu.:5.5   3rd Qu.:8.5
##  Max.   :3.0   Max.   :6.0   Max.   :9.0
```

```r
summary(mtx2)
```

```
##        V1            V2            V3
##  Min.   :1.0   Min.   :2.0   Min.   :3.0
##  1st Qu.:2.5   1st Qu.:3.5   1st Qu.:4.5
##  Median :4.0   Median :5.0   Median :6.0
##  Mean   :4.0   Mean   :5.0   Mean   :6.0
##  3rd Qu.:5.5   3rd Qu.:6.5   3rd Qu.:7.5
##  Max.   :7.0   Max.   :8.0   Max.   :9.0
```

```r
#Further calculations
(mtx3 <- matrix(c(seq(from= -98, to= 100, by=2)), nrow=10, ncol=10))
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  -98  -78  -58  -38  -18    2   22   42   62    82
##  [2,]  -96  -76  -56  -36  -16    4   24   44   64    84
##  [3,]  -94  -74  -54  -34  -14    6   26   46   66    86
##  [4,]  -92  -72  -52  -32  -12    8   28   48   68    88
##  [5,]  -90  -70  -50  -30  -10   10   30   50   70    90
##  [6,]  -88  -68  -48  -28   -8   12   32   52   72    92
##  [7,]  -86  -66  -46  -26   -6   14   34   54   74    94
```

```
##  [8,]  -84  -64  -44  -24   -4   16   36   56   76    96
##  [9,]  -82  -62  -42  -22   -2   18   38   58   78    98
## [10,]  -80  -60  -40  -20    0   20   40   60   80   100
```

```
apply(mtx3,2,mean) #calculating mean of each column, 2 means columns
```

```
##  [1] -89 -69 -49 -29  -9  11  31  51  71  91
```

```
apply(mtx3,1,mean) #calculating mean of each row, 1 means rows
```

```
##  [1] -8 -6 -4 -2  0  2  4  6  8 10
```

```
apply(mtx3,1,sum) #calculating sum of each row, 1 means rows
```

```
##  [1] -80 -60 -40 -20   0  20  40  60  80 100
```

```
apply (mtx3,1,function(x)mean(x[x>0])) #mean of rows of only positive numbers
```

```
##  [1] 42 44 46 48 50 52 54 56 58 60
```

Data Frames

These are two dimensional collection of different data types. The data are organised in unlimited number of rows and columns with each row representing observations and each column representing variables.They are created using the function `data.frame()` to assign an object as a data frame. Each of the columns is vectorised and stores only one type of data. Most of the data collected and used are organised in data frames. To reference a column, use `$` symbol and `[]` for indexing the elements.

```
(df1 <- data.frame(col1 = c(1, 2, 3, 4, 5,7),
                   col2 = c("ann", "john", "erick", "james", "june", "Jasper")))
```

Creating a data frame

```
##   col1   col2
## 1    1    ann
## 2    2   john
## 3    3  erick
## 4    4  james
## 5    5   june
## 6    7 Jasper
```

```
class(df1)
```

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

```
typeof(df1)
```

```
## [1] "list"
```

```
str(df1)
```

```
## 'data.frame':    6 obs. of  2 variables:
##  $ col1: num  1 2 3 4 5 7
##  $ col2: chr  "ann" "john" "erick" "james" ...
```

```
(fname <- c("Ken", "Cate", "Amalia", "Jimmy", "dorcas")) #character vector
```

Create data frames from different types of vectors

```
## [1] "Ken"    "Cate"   "Amalia" "Jimmy"  "dorcas"
```

```
(score <- c(34, 67, 90, 200, 20)) #numeric vector
```

```
## [1]  34  67  90 200  20
```

```
(fcolour = factor(c("blue", "green", "red", "pink", "green"))) #factor vector
```

```
## [1] blue  green red   pink  green
## Levels: blue green pink red
```

```
(gender2 = factor(c("M", "F", "F", "M", "F"))) #factor vector
```

```
## [1] M F F M F
## Levels: F M
```

```
(truefalse = c(TRUE, TRUE, FALSE, TRUE, FALSE))#logical vector
```

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

```
#Combine above to form data frame
(df = data.frame(fname, score, fcolour, gender, truefalse))
```

```
##    fname score fcolour gender truefalse
## 1    Ken    34    blue      F      TRUE
## 2   Cate    67   green      M      TRUE
## 3 Amalia    90     red      F     FALSE
## 4  Jimmy   200    pink      M      TRUE
## 5 dorcas    20   green      M     FALSE
```

```
str(df)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ fname    : chr  "Ken" "Cate" "Amalia" "Jimmy" ...
##  $ score    : num  34 67 90 200 20
##  $ fcolour  : Factor w/ 4 levels "blue","green",..: 1 2 4 3 2
##  $ gender   : chr  "F" "M" "F" "M" ...
##  $ truefalse: logi  TRUE TRUE FALSE TRUE FALSE
```

```
df1[2, 2] #outputs element at row2, column2 from data frame 1=john
```

Retreiving elements of a data frame

```
## [1] "john"
```

```
df1[6, 2]  #outputs element at row6, column 2 =jasper
```

```
## [1] "Jasper"
```

```
df1$col2 #outputs all elements of column2 , $reference the column
```

```
## [1] "ann"    "john"   "erick"  "james"  "june"   "Jasper"
```

```
df1$col2[5] #outputs the element at position 5 of column 2=june
```

```
## [1] "june"
```

```
df1[3, ] #display all elements in row3 -row number
```

```
##   col1  col2
## 3    3 erick
```

```
df1
```

```
##   col1   col2
## 1    1    ann
## 2    2   john
## 3    3  erick
## 4    4  james
## 5    5   june
## 6    7 Jasper
```

```
df1[df1$col3 < 12, ] #display rows with col3 less than 12
```

```
## [1] col1 col2
## <0 rows> (or 0-length row.names)
```

```r
df1[df1$col5 == 1, ] #display rows with col5 equals 1
```

```
## [1] col1 col2
## <0 rows> (or 0-length row.names)
```

```r
df1[df1$col2 == "john", ] #display rows with col2 containing john
```

```
##   col1 col2
## 2    2 john
```

```r
df1[df1$col6 > 1, ] #display rows with col6 greater than 1
```

```
## [1] col1 col2
## <0 rows> (or 0-length row.names)
```

```r
(df1$col3 <- df1$col1*4) #multiplies each element in column 1 by 4 and assigns them to column 3 of the
```

Calculations using elements of a data frame

```
## [1]  4  8 12 16 20 28
```

```r
(df1$col4 <- df1$col1/3) #divides each element of col1 by 3 and outputs in col4
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.3333333
```

```r
df1[, "col4"] #display all elements in col4-col name
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.3333333
```

```r
df1[, 4] #display all elements in col4-col number
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.3333333
```

```r
(df1$col5 <- round(df1$col4, 3)) #rounds off col4 to 4 d.p
```

```
## [1] 0.333 0.667 1.000 1.333 1.667 2.333
```

```r
(df1$col6 <- signif(df1$col4, 3))
```

```
## [1] 0.333 0.667 1.000 1.330 1.670 2.330
```

```r
(df2 <- data.frame(
  var1 = 1:15,
  var2 = 16:30
))
```

Combining data frames

```
##    var1 var2
## 1     1   16
## 2     2   17
## 3     3   18
## 4     4   19
## 5     5   20
## 6     6   21
## 7     7   22
## 8     8   23
## 9     9   24
## 10   10   25
## 11   11   26
## 12   12   27
## 13   13   28
## 14   14   29
## 15   15   30
```

```r
(df3 <- data.frame(
  var3 = 1:15,
  var4 = 16:30
))
```

```
##    var3 var4
## 1     1   16
## 2     2   17
## 3     3   18
## 4     4   19
## 5     5   20
## 6     6   21
## 7     7   22
## 8     8   23
## 9     9   24
## 10   10   25
## 11   11   26
## 12   12   27
## 13   13   28
## 14   14   29
## 15   15   30
```

```r
(df4 <- data.frame(
  var1 = 16:30,
  var2 = 31:45
))
```

```
##    var1 var2
## 1    16   31
## 2    17   32
## 3    18   33
## 4    19   34
## 5    20   35
## 6    21   36
## 7    22   37
## 8    23   38
## 9    24   39
## 10   25   40
## 11   26   41
## 12   27   42
## 13   28   43
## 14   29   44
## 15   30   45
```

```
(df5 <- cbind(df2, df3)) #join two data frames row wise. Must have equal number of rows
```

```
##    var1 var2 var3 var4
## 1     1   16    1   16
## 2     2   17    2   17
## 3     3   18    3   18
## 4     4   19    4   19
## 5     5   20    5   20
## 6     6   21    6   21
## 7     7   22    7   22
## 8     8   23    8   23
## 9     9   24    9   24
## 10   10   25   10   25
## 11   11   26   11   26
## 12   12   27   12   27
## 13   13   28   13   28
## 14   14   29   14   29
## 15   15   30   15   30
```

```
(df6 <- rbind(df2, df4)) #appends a data frame to another, col names must be similar
```

```
##    var1 var2
## 1     1   16
## 2     2   17
## 3     3   18
## 4     4   19
## 5     5   20
## 6     6   21
## 7     7   22
## 8     8   23
## 9     9   24
## 10   10   25
## 11   11   26
## 12   12   27
## 13   13   28
## 14   14   29
```

```
## 15   15   30
## 16   16   31
## 17   17   32
## 18   18   33
## 19   19   34
## 20   20   35
## 21   21   36
## 22   22   37
## 23   23   38
## 24   24   39
## 25   25   40
## 26   26   41
## 27   27   42
## 28   28   43
## 29   29   44
## 30   30   45
```

```
(df2 <- data.frame(
  var1 = 1:15,
  var2 = 16:30
))
```

```
##    var1 var2
## 1     1   16
## 2     2   17
## 3     3   18
## 4     4   19
## 5     5   20
## 6     6   21
## 7     7   22
## 8     8   23
## 9     9   24
## 10   10   25
## 11   11   26
## 12   12   27
## 13   13   28
## 14   14   29
## 15   15   30
```

```
(df2$var3 <- paste(df2$var1, df2$var2)) #concatenates var1 and var2 with space in between
```

```
##  [1] "1 16"  "2 17"  "3 18"  "4 19"  "5 20"  "6 21"  "7 22"  "8 23"  "9 24"
## [10] "10 25" "11 26" "12 27" "13 28" "14 29" "15 30"
```

```
(df2$var4 <- paste0(df2$var1, "-", df2$var2)) #concatenates var1 and var2 with a hyphen and no space be
```

```
##  [1] "1-16"  "2-17"  "3-18"  "4-19"  "5-20"  "6-21"  "7-22"  "8-23"  "9-24"
## [10] "10-25" "11-26" "12-27" "13-28" "14-29" "15-30"
```

Lists

Is one dimensional structures that can hold different types. It's heterogeneous Use the function `list()` to create a list. It's like a bucket of different items put together. Individual elements of a list can be extracted

by referencing the name of object in the list using `$` and by position `x[[i]]`. You can embed vectors, matrices and data frames all in a list.

```
(list0 <- c("look", 1, TRUE, FALSE)) #holds anything parsed to it
```

Creating lists

```
## [1] "look"  "1"     "TRUE"  "FALSE"
```

```
(list1 <- list(
  var1 = 1:20,
  var2 = rep(c("A", "B"), 10)
))
```

```
## $var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $var2
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A"
## [20] "B"
```

```
list1$var2[6]
```

Retreiving elements of a list

```
## [1] "B"
```

```
list1[1]
```

```
## $var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
list1[[1]][6]
```

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

```
list1$var1[6]
```

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

```
(list2 <- list(
  list1 = list(var1 = 1:20,
               var2 = rep(c("A", "B"), 10)),
  var3 = rep(c("A", "B"), 10)
))
```

```
## $list1
## $list1$var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $list1$var2
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B"
## [20] "B"
##
##
## $var3
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B"
## [20] "B"
```

```
class(list2)
```

```
## [1] "list"
```

```
typeof(list2)
```

```
## [1] "list"
```

```
str(list2)
```

```
## List of 2
##  $ list1:List of 2
##   ..$ var1: int [1:20] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ var2: chr [1:20] "A" "B" "A" "B" ...
##  $ var3 : chr [1:20] "A" "B" "A" "B" ...
```

```
list2$list1$var1[1]
```

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

```
list2[[1]][1]
```

```
## $var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
list2$list1$var2[1]
```

```
## [1] "A"
```

```
list2[[2]][1]
```

```
## [1] "A"
```

```
(list_all <- list(
  df1 = list(var1 = 1:20,
             var2 = rep(c("A", "B"), 10)),
  df2 = list(var3 = 1:20,
             var4 = rep(c("A", "B"), 10))
))
```

```
## $df1
## $df1$var1
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $df1$var2
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A"
## [20] "B"
##
##
## $df2
## $df2$var3
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $df2$var4
##  [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A" "B" "A"
## [20] "B"
```

```
#(list_all1 <- do.call(list_all)) #investigate
```

convert list to dataframe

Arrays

Are multidimensional. Can be more than 1 dimensions and as many as user would wish. Can be considered super set of matrices. Are homogeneous. Can be created by combining matrices. You cannot view an array in one go but can access the levels of the arrays (each of the matrices combined). Useful when stacking matrices on top of each other. Arrays cannot be used to perform calculations but to store multiple matrices.

```
v1 =c(1:12)
(m1 = matrix(v1, nrow=3, ncol=4, byrow=TRUE))
```

Creating an array

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

```
v2 = c(25:36)
(m2 = matrix(v2, nrow=3, ncol=4, byrow=TRUE))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   25   26   27   28
## [2,]   29   30   31   32
## [3,]   33   34   35   36
```

```r
(A = array(c(m1,m2), dim=c(3,4,2)))# 3=rows, 4=columns, 2=number of matrices/levels in the array
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   25   26   27   28
## [2,]   29   30   31   32
## [3,]   33   34   35   36
```

```r
A[,,1] #level 1
```

Retreiving levels of an array

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

```r
A[,,2] #level 2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   25   26   27   28
## [2,]   29   30   31   32
## [3,]   33   34   35   36
```

```r
names <- c("jimmy","july", "feb") #row names
info <- c("Height", "Weight", "Hair colour", "skin colour") #column names
(A = array(c(m1,m2), dim=c(3,4,2), dimnames= list(names, info)))
```

Defining array rows and columns

```
## , , 1
##
##       Height Weight Hair colour skin colour
## jimmy      1      2           3           4
## july       5      6           7           8
## feb        9     10          11          12
##
## , , 2
```

```
##
##       Height Weight Hair colour skin colour
## jimmy     25     26          27          28
## july      29     30          31          32
## feb       33     34          35          36
```

```
A[1,,1] #returns first row of first level
```

Retreiving values of an array from different levels

```
##      Height      Weight Hair colour skin colour
##           1           2           3           4
```

```
A[1,,2] #returns first row of second level
```

```
##      Height      Weight Hair colour skin colour
##          25          26          27          28
```

Functions and Loops

Functions are group of instructions that help users perform/complete a specific task. They can be as complex as an `R package` (collection of functions) or as simple as the assign symbol `(<-)`. Most functions have so far written and can be automatically accessed in R while others need to be access via installing R packages but sometimes one needs a customised function to complete a task. Curly brackets `{}` are used to mark the beginning and end of instructions to be performed. The `return()` function is used to explicitly instruct R to print the output/result of the last instruction supplied in a function.

Functions have arguments enclosed in a bracket. The arguments are the inputs that the function would use during processing. There must always be an output in a function sometimes produced by the return function. Functions in R programming are like macros in SAS or STATA programming.

Available Functions

The functions below (not all)are already available can be used to perform specific tasks as desired.

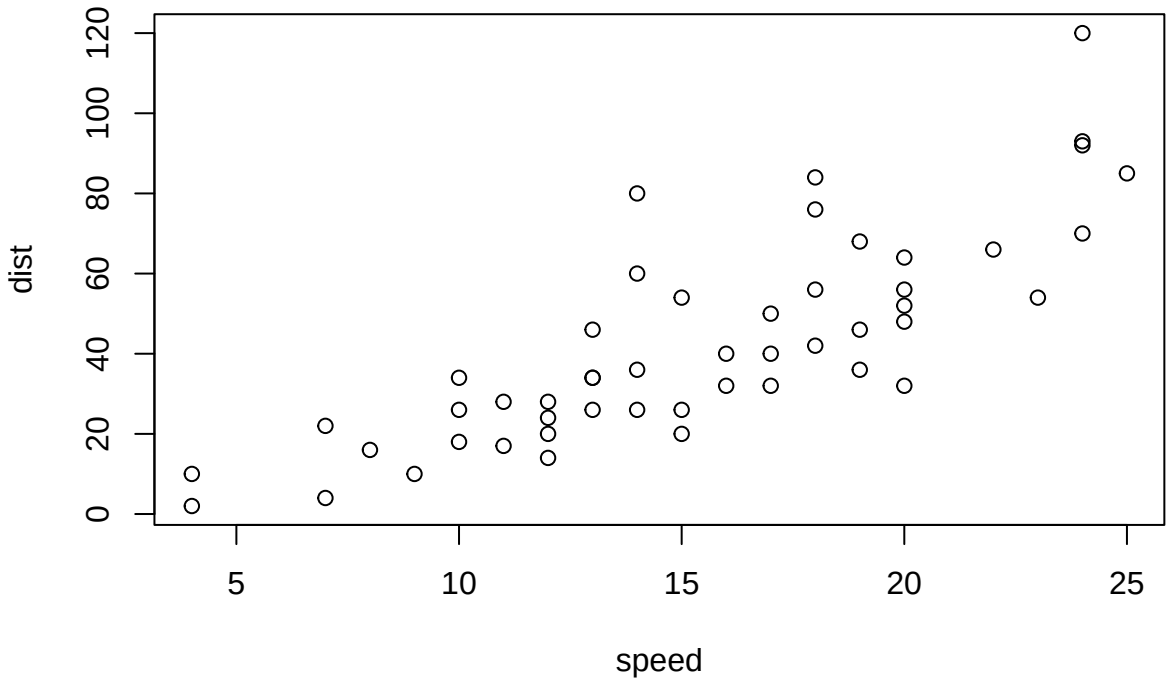
```
v <- c(2,3,45) #assignment symbol for assigning instructions as an object

c(2,3,"x") # c is function that collects together everything in the brackets
```

```
## [1] "2" "3" "x"
```

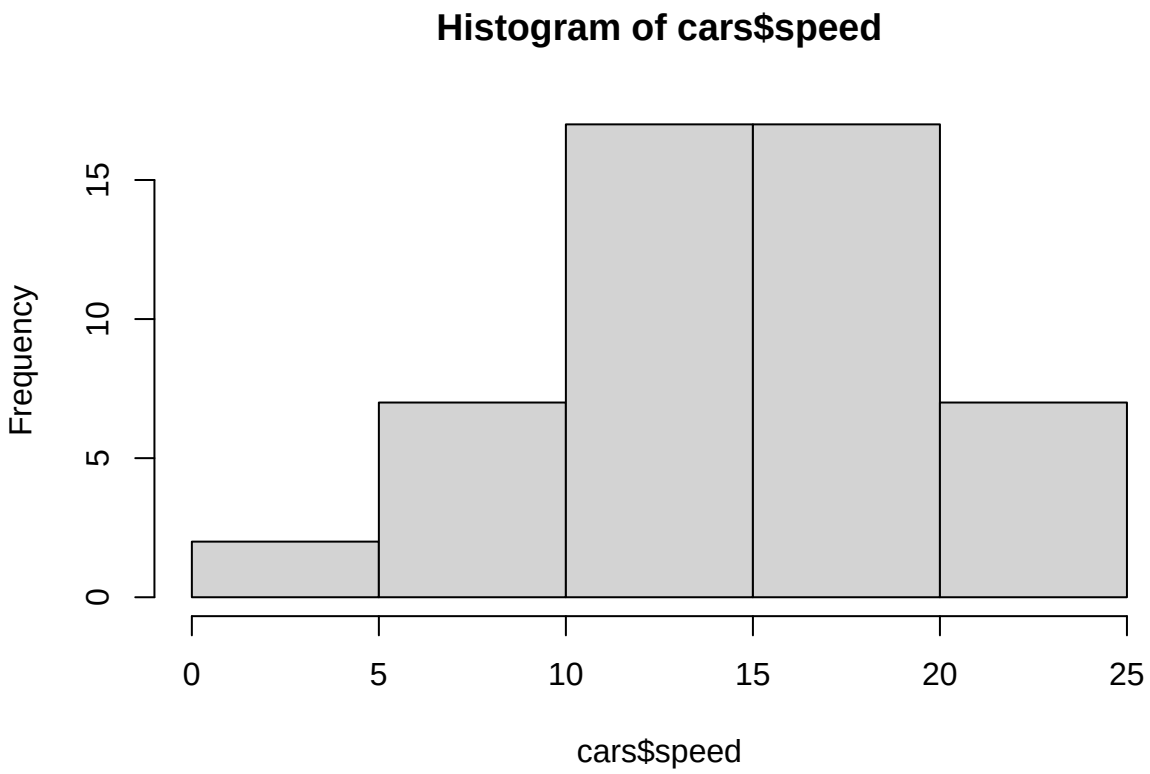
```
sum(2,3,4) #sum functions returns the sum total of items in the bracket
```

```
## [1] 9
```

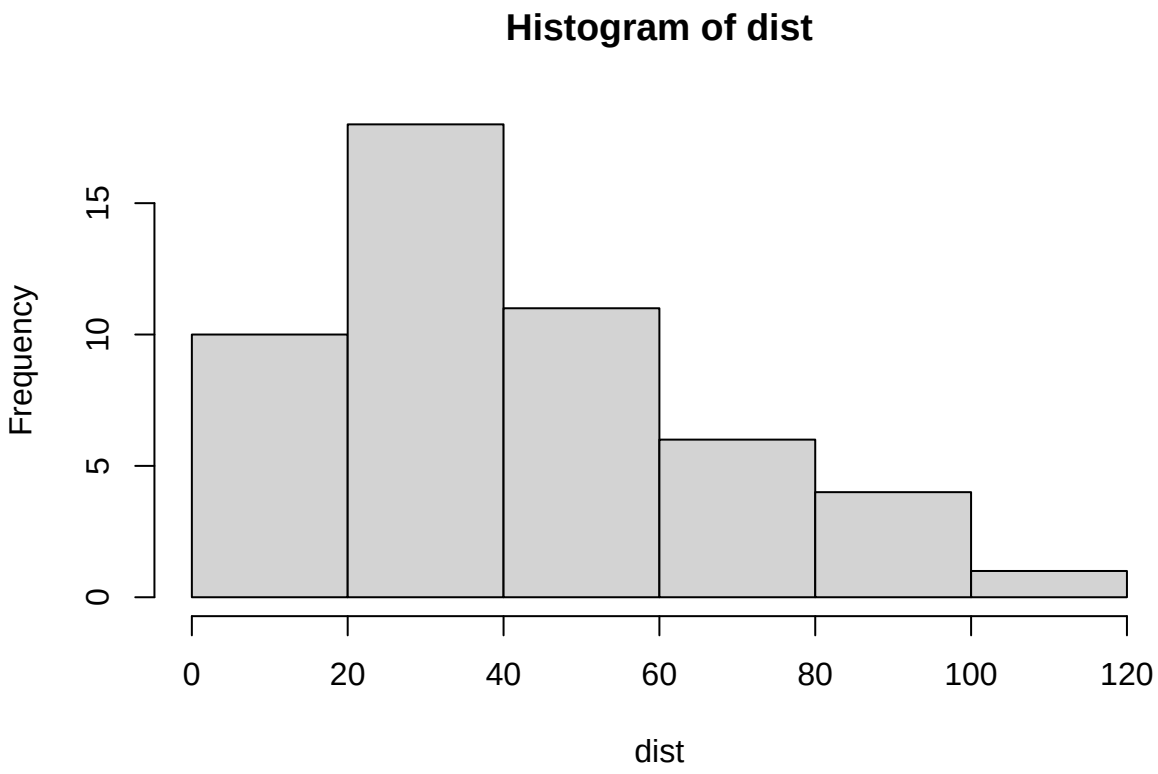
```
plot(cars) #returns a plot of dist vs speed from cars dataset
```




```
hist(cars$speed) #returns a histogram
```




```
attach(cars) # attach function when used before a dataset name, helps user to avoid attaching dataset n

hist(dist) #works without attaching dataset name to the variable
```




```
#View(cars) #enables one view the dataset

head(cars) #returns first six observations
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

```
tail(cars) #returns last six observations
```

```
##    speed dist
## 45    23   54
## 46    24   70
## 47    24   92
## 48    24   93
## 49    24  120
## 50    25   85
```

```
#data() #lists all datasets available in a package

summary(cars) #returns min,1st QTR, median, mean, 3rd QTR, max of all numeric variables in the dataset.
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

```
length(cars) #returns total number of variables in the dataset
```

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

```
length(speed) #returns total number of observations for the variable
```

```
## [1] 50
```

```
unique(dist) #returns all unique entries in a variable
```

```
##  [1]   2  10   4  22  16  18  26  34  17  28  14  20  24  46  36  60  80  54  32
## [20]  40  50  42  56  76  84  68  48  52  64  66  70  92  93 120  85
```

```
cars[10:15,1:2] #[]subsets data to select observations from row 10 to 15 for all the columns
```

```
##    speed dist
## 10    11   17
## 11    11   28
## 12    12   14
## 13    12   20
## 14    12   24
## 15    12   28
```

```
#function(x) #the function itself is a function that instructs R to treat an object as a function for p

#%>% #pipe symbol that represents "THEN" and moves processing moves to the next line of instruction. Is
```

Creating a function

We write functions to help us to write code in easy to understand chunks. Also, it helps to write reusable code avoid repetitive code lines. A functional function must have certain parameters: A function `name` which is the assigned to the function `function()`, `inputs` or arguments supplied inside the brackets (function(x,y,z)), the curly brackets `{}` to enclose the instructions to be performed, `instructions` to performed supplied inside the curly brackets, then finally the `return()` to output the result of the last instruction performed. Below is a general structure of a function.

functionname <- function(inputs){

output_value <- do_something(input)

return(output_value)

}

The curly brackets indicate that the instructions inside them are a group hence must be processed together.

```r
#Function to calculate the volume of a box
    # inputs: length=23, width= 20, height=10

boxvolume <- function(length, width, height){
  area_of_crosssection <- length*height
  volume <- area_of_crosssection * width
  return(volume)
}

#calling the function
boxvolume(23, 20,10)
```

```
## [1] 4600
```

```r
#store the results for use later
bvolume <- boxvolume(23, 20,10)
```

For Loops

For Loops are used for repeating things in R. They are fundamental structures for performing repetitions in programming. They work by performing the same actions for each item in a list of items like a vector. The key word is `for` which means for each item in the list of items supplied. Basic structure looks like this:

for(item in vector_of_items){

do_something(item)

}

You need to first create a vector of items to loop over. There are two variants of the For loops. By value and by index.

For Loops by value For loop by value uses the value of a vector to loop over and over until all the elements of the vector have been processed.

```r
#A For Loop to calculate masses from each volume of the box

volumes <- c(1.6, 3, 8, 9)
for(vol in volumes){
  mass <- 2.65*volumes^0.95
  print(mass)
} #not sure why four rows are created
```

```
## [1]  4.141521  7.525079 19.106510 21.368609
## [1]  4.141521  7.525079 19.106510 21.368609
## [1]  4.141521  7.525079 19.106510 21.368609
## [1]  4.141521  7.525079 19.106510 21.368609
```

For Loops by index For loops by index loops over a vector using the index of each element typically starting at 1. The integers (indices) are used to access the values we need from one or more vectors at the position indicated by the index.

```r
volumes <- c(1.6, 3, 8, 9)

for(i in 1:length(volumes)){
  mass <- 2.65*volumes[i]^0.95
  print(mass)
}
```

```
## [1] 4.141521
## [1] 7.525079
## [1] 19.10651
## [1] 21.36861
```

Looping by index helps in storing the results of the loop for use later unlike looping by values. To store the values of the loop, create an empty object of the same length as the vector of items.

```r
volumes <- c(1.6, 3, 8, 9)
masses <- vector(length=length(volumes), mode="numeric") #vector function creates an empty vector with

for(i in 1:length(volumes)){
  mass <- 2.65*volumes[i]^0.95
  masses[i] <- mass #stores the outputs as it loops
}

masses
```

```
## [1]  4.141521  7.525079 19.106510 21.368609
```

Looping by index also allows users to loop over multiple objects at the same time. For example if you want to change the constant 2.65 and exponent 0.95 depending on the circumstance

```r
volumes <- c(1.6, 3, 8, 9)
cons <- c(2.65, 3.4, 6.1, 4.3)
expo <- c(0.95, 1.1, 1.5, 2.5)

masses <- vector(length=length(volumes), mode="numeric") #vector function creates an empty vector with

for(i in 1:length(volumes)){
  mass <- cons[i]*volumes[i]^expo[i]
  masses[i] <- mass #stores the outputs as it loops
}

masses
```

```
## [1]    4.141521   11.384456  138.027244 1044.900000
```

Looping over files The code below is commented out as there are no files yet in the training folder directory.

```r
#data_files <- list.files(pattern="") #get list of files you want
#results <- vector(length=length(data_files), mode="integer") #create an empty vector of length equal t
```

```
#for(i in 1:length(data_files)){
#  data <- read.csv(data_file[i]) #read the csv file (if they are csv files)
#  count <- nrow(data) #count observations if this is what your want
#  results[1] <- count #store the counts for each data files
#}
```

While Loops

While loops allows you to the code to repeat while a condition is true and stops when the condition is no longer true.

```
#What will be the amount at the end of a year after monthly deposit of 1000 and interest rate of 8% on

Whileloop <- function(){

  acc <- c(0)
  Month = 1

  while (Month<=12){

    acc <- (acc*1.08)+1000 #replace the first vector after each calculation
    print(acc)

    Month <- Month + 1 #replace the counter with the next increment
    #print(counter)
  }
}

Whileloop()
```

```
## [1] 1000
## [1] 2080
## [1] 3246.4
## [1] 4506.112
## [1] 5866.601
## [1] 7335.929
## [1] 8922.803
## [1] 10636.63
## [1] 12487.56
## [1] 14486.56
## [1] 16645.49
## [1] 18977.13
```

```
#Above is similar to For loop but no need of adding the iterator
Forloop <- function(){

  acc=0

  for (Month in 1:12){
    acc <- (acc * 1.08) +1000

    print(acc)
```

```
    #print(Month)
  }

}

Forloop()
```

```
## [1] 1000
## [1] 2080
## [1] 3246.4
## [1] 4506.112
## [1] 5866.601
## [1] 7335.929
## [1] 8922.803
## [1] 10636.63
## [1] 12487.56
## [1] 14486.56
## [1] 16645.49
## [1] 18977.13
```

```
#How do I get just the last value?
```

Looping Using the if Statements

You can combine loops and functions by putting most of the codes we want to run in a function the calling one more functions each time through the loop. As an example we can use a non-vectorised function to estimate the mass. This type of loop uses the `if` and `else` functions.

If statements allows users to write functions that outputs a result if a condition is true.

```
#Example 1 : Combined if statement and For Loop

est_mass <- function(volume){
  if (volume>5){
    mass <- 2.65*volume^0.95
  } else {
    mass <- NA
  }
  return(mass)
}

volumes <- c(1.6,3,8,4,9)
masses <- vector(length=length(volumes), mode="numeric")

for(i in 1:length(volumes)){
  mass <- est_mass(volumes[i])
  masses[i] <- mass
}

masses
```

```
## [1]       NA       NA 19.10651       NA 21.36861
```

```r
(mass_sapply <- sapply(volumes, est_mass)) # this code is similar to the above for loop
```

```
## [1]       NA       NA 19.10651       NA 21.36861
```

```r
#Example 2
#Test what to wear depending on  temperature of the day.

Hoodie <- function(temperature){

  if (temperature <25){
    print("Wear hoodie")
  }
  else if (temperature == 25){
    print("Wear vest")
  }
  else {
    print("Wear nothing")
  }
}

Hoodie(26)
```

```
## [1] "Wear nothing"
```

R Packages

R provides a wide range of packages that enable users to perform specific tasks efficiently in R environment. An R package is a collection of functions and can be downloaded from CRAN (Comprehensive R Archive Network). CRAN is the repository for R packages. Some packages already exist in the R environment but some of them must be installed first before they can used. Some packages exists together in an ecosystem of packages for specific tasks. For example the tydyverse package is an ecosystem of packages that enable users clean data, produce graphs and charts, transform data, import data and handle dates

Installing and loading a package

Installing an R package is very simple. The function `install.packages("packagename")`is used. This makes the package available for access to RStudio but cannot be used yet. One has to load the package into R environment to be able use it and this is done using the function `library(packagename)`. Some packages can exist in more than one ecosystem and one might wish to explicitly indicate the ecosystem from which the package is from by using using double colons as `ecosystemname::packagename`. One has to have an internet connection to be able install packages.

Below are examples of packages commonly used.

```r
install.packages("magrittr")
```

```r
install.packages("table1")
```

```r
install.packages("arsenal")
```

```r
install.packages("tinytex")
```

```r
install.packages("readxl")
```

```r
install.packages("haven")
```

```
install.packages("gmodels")
```

```
library(tidyverse) #en ecosystem (collection) of packages like the dplyr,ggplot2, lubridate, tidyr, tib
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.3     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
library(magrittr)   # package for piping i.e %>% piping is handy when you want to avoid nesting of func
```

```
##
## Attaching package: 'magrittr'
##
## The following object is masked from 'package:purrr':
##
##     set_names
##
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
library(table1) #for descriptive data analysis in HTML
```

```
##
## Attaching package: 'table1'
##
## The following objects are masked from 'package:base':
##
##     units, units<-
```

```
library(arsenal) #for running proc compare on two datasets and other complex tasks
```

```
##
## Attaching package: 'arsenal'
##
## The following object is masked from 'package:magrittr':
##
##     set_attr
##
## The following object is masked from 'package:lubridate':
##
##     is.Date
```

```
library(tinytex) #contains functions for compiling LaTex documents

library(readxl) #for importing xls dataset

library(haven) #for importing and exporting SPSS, STATA and SAS files

library(gmodels) #package for model fitting

library(knitr) #package for dynamic reporting in R
```

Data Wrangling

Data wrangling includes a series of processes that involves tidying data that ends up with suitable datasets for analysis. This starts from reading in data from where it's stored and having a clean data after all the processes. The tidyverse ecosystem of packages has packages of functions that support data wrangling. Package for reading in data `readr`, for generating visuals `ggplot2`, for tidying data `tidyr`, for formatting dates `lubridate`, for modifying a data frame for efficient reading `tibble` and for transforming data `dplyr`. These packages have functions inside them that support what they are intended for.

Setting work directory

You need to know your working directory before reading in directory. To know the current working directory use the function `getwd()`. If the directory is different from where you need to save your work, you need to change/set your work directory using the function `setwd()` copy the path to where you want to save work the paste in the setwd() function all wrapped in `""`. Change the backslashes `\` to forward slashes `/` in the path directory.

Use `dir()` to view the files in the directory.

```
#Knowing your directory
getwd()
```

```
## [1] "C:/Users/Dwayne/OneDrive/R/Projects/RTraining"
```

```
dir() #get list of files in the
```

```
## [1] "Cleaning.R"      "Dataset"         "LearningR.html"  "LearningR.Rmd"
## [5] "LearningR.tex"   "LearningR_files" "LICENSE"         "README.md"
## [9] "RTraining.Rproj"
```

```
#You can save the list of the files in a R object in case you want to use the list.

(file <-dir())
```

```
## [1] "Cleaning.R"      "Dataset"         "LearningR.html"  "LearningR.Rmd"
## [5] "LearningR.tex"   "LearningR_files" "LICENSE"         "README.md"
## [9] "RTraining.Rproj"
```

```
list.files() #an alternative to dir()
```

```
## [1] "Cleaning.R"      "Dataset"         "LearningR.html"  "LearningR.Rmd"
## [5] "LearningR.tex"   "LearningR_files" "LICENSE"         "README.md"
## [9] "RTraining.Rproj"
```

Reading and writing datasets

The readr package of the tidyverse has functions that enables R users to read and write/save files of different formats.

`CSV files` are text files that have data/values separated by commas. The `read.csv(csvfilename)` functions from readr reads in the csv file and converts it to a data frame for use in R. The `write.csv` function saves he csv file into the working directory. Run `?read.csv` or `?write.csv` to check for the options allowed for these functions.

`Excel` files can include `xls` or `xlsx`. These are files with data arrange in rows and columns. R has the package `readxl` that allows user to read in these files. Use the function `read_excel(excelname)` to read the files. `read_xls()` and `read_xlsx()` also exists for reading excel files. Run `?read_excel` to learn about the options that the function can take.

```
(df <- read_excel("Dataset/STIData.xls")) #practice data for this training
```

```
## New names:
## * 'Sex' -> 'Sex...35'
## * 'Sex' -> 'Sex...47'
```

```
## # A tibble: 227 x 47
##    IdNumber CaseStatus Date                A1Age A2Occupation A3Church
##       <dbl>      <dbl> <dttm>              <dbl> <chr>        <chr>
##  1       32          2 2009-12-03 00:00:00    23 1 unemployed 5 pentecostal
##  2       33          1 2009-12-03 00:00:00    24 4 student    2 apostolic
##  3       34          2 2009-12-03 00:00:00    24 1 unemployed 2 apostolic
##  4       35          1 2009-12-03 00:00:00    33 3 formal     7 roman catholic
##  5       10          2 2009-12-03 00:00:00    63 4 student    8 other
##  6       11          1 2009-12-03 00:00:00    22 3 formal     6 atheist
##  7       12          2 2009-12-04 00:00:00    22 2 informal   6 atheist
##  8       13          1 2009-12-04 00:00:00    19 4 student    7 roman catholic
##  9       14          2 2009-12-04 00:00:00    22 3 formal     5 pentecostal
## 10       15          1 2009-12-04 00:00:00    29 3 formal     6 atheist
## # i 217 more rows
## # i 41 more variables: A4LevelOfEducation <chr>, A5MaritalStatus <chr>,
## #   Weight <dbl>, Height <dbl>, C3StiYesno <dbl>, D1BurialSociety <dbl>,
## #   D1religiousgrp <dbl>, D1savingsClub <dbl>, D1tradersAssoc <dbl>,
## #   D2Group1 <chr>, D2Group2 <chr>, D3Education <dbl>,
## #   D3FuneralAssistance <dbl>, D3HealthServices <dbl>, DurationOfillness <dbl>,
## #   E8WhyhaveSTI <chr>, N10givereceiveforsex <chr>, N11Usedcondom <chr>, ...
```

```
#View(df) #you can view the data to see all the variables and observations.
```

Reading and writing txt files (.txt/.doc)

Reading and writing SAS, STATA, SPSS Files

Reading tables/databases

Web Scrapping

Exploring data

Cleaning data

Handling Missing data

Handling dates and times

Handling strings

Handling dupicates

Handling factor datapoints

Transforming data

Exploratory Data Analysis

Univariate data analysis

Multivariate Data analysis

Data Visualisation

Boxplots

Piecharts

Histograms

Barcharts

Heat maps

Bubble graphs

R Shiny

Building and interactive dashboard

Developing an RShiny Web App

ADVANCED PROGRAMMING