Module 1

Overview and History of R

R is a dialect of the S language.

What is S?

S is a language that was developed by John Chambers and others at Bell Labs. S was initiated in 1976 as an internal statistical analysis environment - originally implemented as Fortran libraries. Early versions of the language did not contain functions for statistical modeling. In 1988 the system was rewritten in C and began to resemble the system we see today. In 1993, Bell Labs gave StatSci an exclusive license to develop and sell the S language. In 2004 Insightful purchased the S language from Lucent and is the current owner. In 2006 Alcatel purchased Lucent technologies. Insightful sells its implementation of S under the name S-PLUS. In 2008, Insightful is acquired by TIBCO for $25 million. The makers of S wanted to create an interactive environment for users to work with data and perform basic analyses. Later the user could use the languahe to develop their own tools.

A short history of R

R was created in 1991 by Ross Ihaka and Robert Gentleman. In 1993, R was announced to the public for the first time. In 1995, Martin Machler convinced Ross and Robert to use the GNU General Public License; thus making it a free software. In 1996, a mailing list was created for developers using R. This led to the formation of the R core group in 1997; this group controls the source code of R

Features of R

- Syntax is very similar to S, making migration easier.
- Semantics are superficially similar to S, but in reality are quite different.
- Can run on any standard computing platform or operating system.
- There is active development with frequent bugfix releases.
- The software itself is quite lean; the functionalities are organized in modular packages
- It has much better graphic and visualization capabilities when compared to most other stat packages.
- While it is useful for working interactively (Markdown and Console), it is also a powerful programming language to develop new tools
- It has a vibrant and active user community.
- It is free!!

What is meant by "free software"?

Free software, libre software, libreware sometimes known as freedom-respecting software is computer software distributed under terms that allow users to run the software for any purpose as well as to study, change, and distribute it and any adapted versions.Free software is a matter of liberty, not price; all users are legally free to do what they want with their copies of free software (including profiting from them) regardless of how much is paid to obtain the program.

- The freedom to run the program, for any purpose.
- The freedom to study how the program works; adapt it to your needs. Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbour.
- The freedom to improve the program, and release your improvements to the public so that the whole community benefits.

Drawbacks of R

- Essentially based on a 40 year old technology.
- Minimal built-in support for dynamic or 3D graphics.
- New functionality is based on consumer demand and user contributions.
- Objects must be stored in physical memory; however there are methods to deal with this.
- Not ideal for all situations.

Design of the R system

The R system divided into two conceptual parts: base R and everything else. R functionality is divided into a number of packages.

- The "base" R system contains the **base** package required to run R, and contains the most fundamental actions
- The other packages contained in the base system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.
- There are also recommended packages like boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nime, rpart, survival, MASS, spatial, nnet, Matrix.

There are about 4000 packages on CRAN that have been developed by users and programmers around the world. There are also many packages associated with the Bioconductor project. Additionally, a lot of people also make packages available on their personal websites; there is no reliable way to keep track of how many packages exist in this way.

R console input and evaluation

Expressions can be typed into the R prompt for evaluation. The <- operator is used to assign a value to a variable

```r
x <- 1
x
```

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

```r
msg <-"Hello"
```

Entering the name of a variable in the console will display its value. One can also use the print( ) function.

```r
print(x)
```

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

When a complete expression is entered at the prompt, it is evaluated and the result of the expression is returned. It may sometimes be auto-printed (e.g. when entering a variable, we get its value).

```r
msg
```

```
## [1] "Hello"
```

[1] indicates that msg has one element and that it is the string Hello.

The colon is used to create a sequence of integers.

```r
x <- 6:25
x
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

The result has [1] before printing the elements.

Data types in R

Objects and Attributes

R has five basic categories of objects: Character, numeric, integer, complex, logical. The most basic object in R is a vector. A vector can only contain objects of the same class. The only exception is a list; which can be a collection of objects belonging to different classes.

Numbers in R are generally treated as numeric objects i.e. double precision real numbers. If you need an integer, specify the L suffix. 1 will return a numeric type while 1L will return an integer type. R also lets you use a special number called Inf. It can be used in regular calculations. 1/0 will return Inf while 1/Inf will return 0. NaN stands for undefined values. 0/0 will return NaN.

Objects in R can have attributes associated with them. Attributes are: names or dimnames, dimensions (for arrays and matrices), class, length and any other attribute that is user-defined. Attributes of an object can be accessed using the attributes( ) function.

Vectors and Lists

The c( ) function is used to create vectors of objects. It concatenates the elements together.

```r
x <- c(0.5,0.6)          ##numeric
x <- c(TRUE, FALSE)      ##logical
x <- c(T,F)              ##logical
x <- c("a", "b", "c")    ##character
x <- 9:29                ##integer
x <- c(1+0i, 2+4i)       ##complex
```

We can also use the vector( ) function to create a vector of a certain type and length. In the example below we see a numeric vector of size 10. It will be initialized with the default value of its type. For numeric the default is 0.

```r
x <- vector("numeric", length = 10)
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Apart from the : operator, the seq function can also be used to create sequences of numbers.

```r
seq(0,10, by=0.5)
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
## [16]  7.5  8.0  8.5  9.0  9.5 10.0
```

The by argument is used to specify the value by which we should increment / decrement from one element to the next. So the code above gives a vector that starts at 0 and goes up to 10, increasing by 0.5 each time. Sometimes, we may only want a certain number of elements, regardless of the increment. The code below gives a vector that starts from 5 (first element is 5) and ends at 10 with length 30 (thirtieth element is 10). It calculates the appropriate value to increment by in order to satisfy this condition.

```r
seq(5,10, length=30)
```

```
##  [1]  5.000000  5.172414  5.344828  5.517241  5.689655  5.862069  6.034483
##  [8]  6.206897  6.379310  6.551724  6.724138  6.896552  7.068966  7.241379
## [15]  7.413793  7.586207  7.758621  7.931034  8.103448  8.275862  8.448276
## [22]  8.620690  8.793103  8.965517  9.137931  9.310345  9.482759  9.655172
## [29]  9.827586 10.000000
```

```r
my_seq <- 10:15   #This is a vector having 6 elements

seq_along(my_seq)
```

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

seq_along is a handy function when you want to create a vector starting from 1 upto a certain length. In the above code, we can see it creates a vector of the exact same size as the argument that it gets passed.

When you mix different types of objects in a single vector, the general rule is that R will try to coerce the vector to get each element into the same class. This is called coercion.

```r
y <- c(1.7, "a")
y
```

```
## [1] "1.7" "a"
```

```r
z <- c(TRUE, 3)
z
```

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

```
x <- c("a", FALSE)
x
```

```
## [1] "a"     "FALSE"
```

We can see in the first example that the resultant vector is a vector of strings. This is because R can convert 1.7 into character but not convert "a" into a number. In the second example, the logical value TRUE is converted to the number 1; FALSE would have been converted to zero. In the third example, the logical value FALSE is converted to a string.

Logical Vectors

```
num_vect <- c(0.5, 55, -10, 6)
tf <- num_vect<1
```

This created a vector of numbers and then a second vector with the result of the logical expression <1. So the result we get is a logical vector, with each element being either TRUE or FALSE. The value depends on the result of the expression.

```
num_vect >= 6
```

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

Character vectors

```
my_char <- c("My", "name", "is", "Stewart")
my_char
```

```
## [1] "My"      "name"    "is"      "Stewart"
```

```
paste(my_char, collapse =" ")
```

```
## [1] "My name is Stewart"
```

The collapse argument to the paste() function tells R that when we join the elements of the my_char vector, they should be separated with single spaces. collapse is used when we have a single vector; whose elements need to be joined. sep is used when we have multiple vectors that need to be joined.

Classes and coercion

```
x <- 0:6
x
```

```
## [1] 0 1 2 3 4 5 6
```

```r
class(x)
```

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

```r
as.logical(x)
```

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

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

Explicit coercion can be done using the as.* functions in R.

```r
x <- c("A", "B", "C")
as.numeric(x)
```

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

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

```r
as.logical(x)
```

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

If a conversion is attempted that is not possible, we get NaN and corresponding warnings.

Lists are a special type of vector that can contain elements of different classes

```r
x <- list(1, "a", TRUE, 1+2i, FALSE)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+2i
##
## [[5]]
## [1] FALSE
```

Here you can see that the first element is a vector of integers, the second is a vector of characters, the third is a vector of logical TRUE and so on. Elements of a list will have double brackets around them.

```r
x <- list("a", "b", 0.5, 1, FALSE, "d")
x
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
##
## [[3]]
## [1] 0.5
##
## [[4]]
## [1] 1
##
## [[5]]
## [1] FALSE
##
## [[6]]
## [1] "d"
```

Matrices

Matrices are vectors with a dimension attribute. The dimension attribute is an integer vector of length 2 (nrow, ncol)

```r
m <- matrix(nrow = 2, ncol = 3)
m
```

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

```r
dim(m)
```

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

```r
attributes(m)
```

```
## $dim
## [1] 2 3
```

Matrices are constructed column-wise, so entries can be thought of starting in the upper-left corner and running down the columns.

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

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

Matrices can also be created by column-binding or row-binding with cbind( ) and rbind( )

```r
x <- 1:3
y <- 10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Factors

Factors are used to represent categorical data. Factors can be unordered or ordered. Gender (male, female) would be an unordered factor wheras something like a grade (A, B, C, D) is ordered because it has a rank. One can think of of a factor as an integer vector where each integer has a label. Factors are treated specially by modeling functions like lm( ) and glm( ). Using factors with labels is better than using integers because factors are self-describing: having a variable with values "male" and "female" is better than a variable with values 1 and 2.

A factor can be created using the factor( ) function in R. Its input should usually be a character vector

```r
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

The function table will give a frequency count of how many elements of each level are present in the factor.

```r
table(x)
```

```
## x
##  no yes
##   2   3
```

Unclass reduces the factor to an integer vector. In the example below, yes is coded as 2 and no is coded as 1. A factor at its core is an integer vector with the levels attribute.

```r
unclass(x)
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

The order of the levels of a factor can be set by using the levels( ) argument to a factor( ). This can be important in linear modeling because the first level is used as the baseline level. If levels are not mentioned then R processes them alphabetically. So because N comes before Y, no would be the baseline level and yes will be the second level

```r
x <- factor(c("yes", "yes", "no", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no  yes no
## Levels: yes no
```

In the factor initialized above, yes is the baseline level and no is the second level. It can be verified using the unclass function.

```r
unclass(x)
```

```
## [1] 1 1 2 1 2 1 2
## attr(,"levels")
## [1] "yes" "no"
```

Missing values

Missing values in R are denoted by either NA or NaN. NaN is used for undefined mathematical operations. And NA is pretty much used for everything else. There's a function in R called is.na and is.nan which is used to test objects to see if there are missing values in that object. NA values also have a class, so there can be integer NA, character NA, etc. A NaN value is also NA but the converse is not true

```r
X <- c(1, 2, NA, 10, 3)
is.na(X)
```

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

```r
is.nan(X)
```

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

The example below illustrates how NaN is considered to be NA, but NA is not a NaN

```r
Y <- c(1, 2, NA, NaN, 4)
is.na(Y)
```

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

```r
is.nan(Y)
```

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

```r
y <- rnorm(1000)
z <- rep(NA,1000)
my_data <- sample(c(y,z), 100)
my_na <- is.na(my_data)
my_na
```

```
##   [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [13] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE
##  [37]  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
##  [49]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
##  [61]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [73]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [85]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
##  [97] FALSE  TRUE  TRUE  TRUE
```

Trying to use the == operator to check for NAs

```r
my_data == NA
```

```
##   [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [26] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [51] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
##  [76] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

The reason you got a vector of all NAs is that NA is not really a value, but just a placeholder for a quantity that is not available. Therefore the logical expression is incomplete and R has no choice but to return a vector of the same length as my_data that contains all NAs.

To calculate how many NAs in my_data : The trick is to recognize that underneath the surface, R represents TRUE as the number 1 and FALSE as the number 0. Therefore, if we take the sum of a bunch of TRUEs and FALSEs, we get the total number of TRUEs.

```r
sum(my_na)
```

```
## [1] 45
```

```r
my_data
```

```
##   [1]          NA          NA          NA          NA          NA  1.18746509
##   [7]          NA  0.58087726 -2.69130756  2.03771520  0.75152584 -0.56844481
##  [13] -0.60427981 -0.87988629          NA  0.41187115          NA  0.70453882
##  [19] -0.04533295 -1.43451584 -0.40542771 -0.63874218  0.62011237 -0.05156722
##  [25] -1.70183040  0.56598655 -0.20228749  0.68781219 -1.96582526  0.16235815
##  [31]          NA -1.93757733          NA          NA  0.53952122 -0.12811343
##  [37]          NA          NA -0.66540517          NA          NA -0.15340315
##  [43] -0.03407065 -0.97769031 -0.01226109 -0.87994214          NA  0.52748253
##  [49]          NA          NA  0.29227687  1.35296788          NA  0.07058966
##  [55]          NA -0.64519280 -1.48044779          NA          NA -0.40167494
##  [61]          NA  0.33721913  0.51467733  1.37806979 -0.56129525          NA
##  [67]  0.92928289          NA          NA          NA          NA          NA
##  [73]          NA          NA          NA -0.72667592          NA          NA
##  [79] -1.32366898 -0.43366356          NA  0.80410150 -0.05707631  0.48956739
##  [85]          NA          NA -0.82074878 -1.27270507 -0.94778027          NA
##  [91] -0.77601507          NA          NA -0.36087151          NA          NA
##  [97] -0.80151670          NA          NA          NA
```

Data Frames

Data frames are used to store tabular data. They are represented as a special type of list where every element of the list has the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. Unlike matrices, data frames can store different classes of objects in each column. Data frames also have a special attribute called row.names( )

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

Names

R objects can also have names. This can be used for writing readable code and self-describing objects.

```
x <- 1:3
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

Lists can also have names

```
x <- list(a=1, b=2, c=3)
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

Names for matrices can be set as follows. Provide a list with two vectors, one for the rows and one for columns.

```
m <- matrix(1:4, nrow=2, ncol=2)
dimnames(m) <- list(c("a", "b"), c("c", "d"))
m
```

```
##   c d
## a 1 3
## b 2 4
```

We can also convert a vector into a matrix by setting dimensions for it

```
my_vector <- 1:20
my_vector
```

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

```
dim(my_vector)
```

```
## NULL
```

```
dim(my_vector) <- c(4,5)
my_vector
```

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

```
class(my_vector)
```

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

Reading and Writing Data in R: The basics

The read.table() and read.csv() functions are used to read tabular data. They are the most commonly used functions for reading data into R. They can read any kind of files that are formatted as rows and columns and return a data frame in R. The function readLines() is for reading lines of a text file; this returns text as a character vector in R. The function source() is used for reading R code. The dget() function is also used for reading R code files, specifically for R objects that have been deparsed into text files. The unserialize() function is used to read R objects in binary form. The analogous functions for writing data are write.table(), writeLines(), dump(), dput(), save() and serialize().

Reading Tabular data

The read.table function is the most commonly used for reading data into R. It has some important arguments.

- file: the name of a file, or a connection
- header: logical, indicating if the file has a header line
- sep: a string, indicating how the columns are separated
- colClasses: a character vector indicating the class of each column in the dataset
- nrows: the number of rows in the dataset
- comment.char: a character string indicating the comment character
- skip: integer, the number of lines to skip from the begeinning
- stringAsFactors: logical, should the character variables be coded as factors? It defaults to true,

For small to moderately sized datasets, you can usually call read.table() without specifying any other arguments. R will automatically skip lines that begin with a #, figure our how many rows are there (and how much memory needs to be allocated), figure what type of variable is in each column of the table. However, telling R all of these things directly makes it run faster and more efficiently. This is why its important to specify these arguments when the dataset is large. read.csv is identical to read.table except that the default separator is a comma. For read.table the default separator is a single space.

Reading large tables

With much larger datasets, some of the following points can help to ensure quicker execution. Providing as much information to R as possible by providing the arguments mentioned above. Additionally, making a rough calculation of the memory required to store the dataset. if there is not enough RAM, it is best to not execute it. Set comment.char = “ ” if there are no comments in the data file. Setting nrows does not make R run faster but helps with memory usage; its alright to do a mild overestimate. A The colClasses argument can make read.table() much faster. In order to use this option, it is necessary to know the class of each column in the data. A quick way to figure out classes is the following:

```
initial <- read.table("datatable.txt", nrows=100)  #read only a small number of rows
classes <- sapply(initial, class)                  #ascertain the classes
tabAll <- read.table("datatable.txt", colClasses=classes)
```

In general, when using R it is useful to know some things about the system itself. How much memory and RAM is available? What other applications are in use? Are there other users logged in to the system? What is the operating system? Is the OS 32 bit or 64 bit?

A rough calculation of Memory requirements: I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly how much memory is required to store this data frame?

The memory required to store this data frame is approximately:

$1,500,000 \times 120 \times 8$ bytes/numeric $= 1,440,000,000$ bytes

$\frac{1,440,000,000}{1024^2} \approx 1,373.29$ MB

$\frac{1,373.29}{1024} \approx 1.34$ GB

Textual data formats

The functions dump() and dput() are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable. Unlike writing out a table or a CSV file, dump and dput preserve the metadata (sacrificing some of the readability), so that other users do not need to specify it all over again.

Data in a tectual foramt works much better with version-tracking systems like SVN and Git as compared to csv or tsv files. One of the downsides of this format is that it is not very space-efficient.

The dput() function takes an arbitrary R object, it will take most types of R objects and will create some R code that will essentially reconstruct the object in R.

```r
y <- data.frame(a=1, b="a")
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

Often, printing the result of a dput is not very useful. Its more common to save the result of dput into an R file so it can be executed to generate the object later. For deparsing multiple R objects at once, we can use the dump() function. You pass a character vector which contains the names of the objects.

```r
x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
```

The dump is saved in data.R. Now we can delete the objects and then regenerate them using the result of the dump.

```r
rm(x,y)            #code to delete objects
source("data.R")
y
```

```
##   a b
## 1 1 a
```

Connections : Interfaces with the outside world

Data are read in using connection interfaces. Connections can be made to files or to other more exotic things. The functions file, gzfile (file compressed with gzip), bzfile (file compressed with bzip2), url (webpage) are some common connections.

Some of the arguments of the file function are as follows:

- description : Name of the file
- open: a code indicating "r" for read-only, "w" for writing (and initializing new files), "a" for appending, "rb", "wb" and "ab" to read, write or append in binary mode
- blocking: If set to TRUE, then functions using the connection do not return to the R evaluator until the read/write is complete. In non-blocking mode, operations return as soon as possible, so on input they will return with whatever input is available (possibly none) and for output they will return whether or not the write succeeded.

In general, connections are powerful tools that let you navigate files or other external objects. In practicem we don't often deal with the connection interface directly.

```r
con <- file("foo.txt", "r")
data <- read.csv(con)
```

```
## Warning in read.table(file = file, header = header, sep = sep, quote = quote, :
## incomplete final line found by readTableHeader on 'foo.txt'
```

```
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

```
## Warning in read.table(file = file, header = header, sep = sep, quote = quote, :
## incomplete final line found by readTableHeader on 'foo.txt'
```

The writeLines function takes a character vector and writes each element one line at a time. To read lines from files or webpages, we can use the readLines function

```
con <- url("https://theconversation.com/what-owning-a-cat-does-to-your-brain-and-theirs-264396", "r")
x <- readLines(con)
head(x)
```

```
## [1] ""
## [2] "<!DOCTYPE html>"
## [3] "<html lang=\"en-GLOBAL\" class=\"no-js\">"
## [4] "<head>"
## [5] "  "
## [6] "  <meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">"
```

Subsetting

There are a number of operators that can be used to extract subsets of R objects. The [always returns an object of the same class as the original; it can be used to select more than one element. The [[is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame. The $ is used to extract elements of a list or data frame by name; the semantics are same as the [[.

```
x <- c("a", "b", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```
x[2:4]
```

```
## [1] "b" "c" "d"
```

This was an example of subsetting the vector x using a numeric index. We can also do subsetting with a logical index.

```
x[x>"b"]
```

```
## [1] "c" "d"
```

Here, the comparison is based on lexicographical ordering. Letters greater than "b" means all letters from c through to z; the subset is formed by only those elements that return TRUE based on this condition.

Subsetting - Lists

```
x <- list(foo=1:4, bar=0.6)
x[1]
```

```
## $foo
## [1] 1 2 3 4
```

Because the single square bracket will always return the same class. So we get a list with an element named foo. foo contains the sequence 1 to 4.

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

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

When using the double brackets, we don't get a list but just the sequence 1 through 4.

```
x$bar
```

```
## [1] 0.6
```

```
x$foo
```

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

The dollar sign gives us the element associated with the given name. The advantage with this method is you do not need to know the location (index) of the element.

To extract multiple elements of a list, the single bracket operator is used. The double bracket and dollar sign can only return a single list element.

```
x <- list(foo=1:4, bar=0.6, baz="hello")
x[c(1,3)]
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

The [[operator can be used with computed indices, while $ can only be used with literal names.

```
x <- list(foo=1:5, bar=2.6, baz="hello world")
name <- "foo"
x[[name]]
```

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

```r
x$name
```

```
## NULL
```

For subsetting nested elements in a list, one can use the [[operator.

```r
x <- list(a = list(10, TRUE, 14.5), b=c(3.14, 2.81))
x[[1]][[3]]
```

```
## [1] 14.5
```

```r
x[[c(2,1)]]
```

```
## [1] 3.14
```

This would return the third element of the first element of x and first element of the second element of x.

Subsetting - Matrices

Matrices can be subsetted using a pair of indices. The first index will be the row index and the second one will be the column index. You don't meed to always specify both indices of a matrix.

```r
x <- matrix(1:6, 2,3)
x
```

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

```r
x[1,2]        #first row second element
```

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

```r
x[2, ]        #entire second row
```

```
## [1] 2 4 6
```

```r
x[ ,3]        #entire third column
```

```
## [1] 5 6
```

By default, when a single element of a matrix is retrieved, we get a vector of length 1 rather than a 1x1 natrix. This behaviour can be turned off by setting drop=FALSE. Similatly, when subsetting a single column or a single row, we get a vector rather than a matrix. Depending on what is the required output, we can set the value of the drop argument

```r
x[1,2, drop=FALSE]
```

```
##      [,1]
## [1,]    3
```

```r
x[1, , drop=FALSE]
```

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

Subsetting - Partial matching

Partial matching of names is allowed with the [[and $

```r
x <- list(aardvark = 1:5)
x$aa
```

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

```r
x[["a", exact = FALSE]]
```

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

Subsetting - Removing missing values

```r
x <- c(1, 2, NA, 4, NA, 6)
```

One method is to create a logical vector which tells you where the NA's are and so that you can remove them by sub-setting.

```r
bad <- is.na(x)
x[!bad]
```

```
## [1] 1 2 4 6
```

When there are multiple things (lists / vectors / matrices) and we want to clean them up:

```r
x <- c(1, 2, NA, 4, NA, 5)
y <-c("a", "b", NA, "c", NA, "d")
good <- complete.cases(x,y)
good
```

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

```
x[good]
```

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

```
y[good]
```

```
## [1] "a" "b" "c" "d"
```

If both x and y have missing values at different places, good will return TRUE only for those positions that had valid values in both vectors

```
x <- c(1, 2, NA, 4, NA, 5)
y <-c("a", NA, "b", "c", NA, "d")
good <- complete.cases(x,y)
x[good]
```

```
## [1] 1 4 5
```

```
y[good]
```

```
## [1] "a" "c" "d"
```

Vectorized Operations

Vectorized operations is one of the features of the R language that makes it easy to use on the command line without needing to do lots of looping. Many operations in R are vectorized making the code more efficient, concise, and easier to read. It allows computation in parallel

```
x <- 1:4; y <- 6:9
x + y
```

```
## [1]  7  9 11 13
```

```
x * y
```

```
## [1]  6 14 24 36
```

Simply doing x + y will add each element of x with its corresponding element of y. If vectorized operations were not available, this would need a loop to go through x[1] + y[1] then x[2] + y[2] and so on. In R we can use the mathematical operators on vectors directly.

```
x > 2
```

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

```r
y == 8
```

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

We can also use logical operators in vectorized operations.

Vectorized Matrix Operations

```r
x <- matrix(1:4, 2, 2)
y <- matrix(rep(10,4), 2, 2)     #rep means repeat 10 four times
x
```

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

```r
y
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
```

Lets try multiplying these two matrices

```r
x * y
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

The result gave us an element-by-element multiplication. For true matrix multiplication, use % before and after the * operator.

```r
x %*% y
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```