1

STAT2008/STAT6038

Introduction to R

Introduction to R

2

- The R environment allows us to:
  - Manipulate data
  - Use graphics to analyse data
  - Perform calculations
  - Perform statistical analysis
  - Very similar to the programming language called "S", "S-Plus", "S+" which is used in the brick

Introduction to R

3

- You cannot learn R through attending lectures or reading lecture material alone
- The majority of your learning will come from writing R code yourself, making mistakes and learning from those mistakes
- R is something that you will have to learn during tutorials and in your own time

R Workspace

4

- **Entering commands**
  - commands and assignments executed or evaluated immediately
  - separated by new line (Enter/Return) or semicolon
  - recall commands with ↑ or ↓
  - case sensitive
  - every command is some sort of function that does something
  - data and functions are both objects stored in R

R Language

5

- R is an interactive system.
- You type a command, and R returns an answer.
- **Commands** are either:
  - **Expressions.**

    An expression is a sequence of symbols interpreted by R.
  - **Assignments**(of a value to a name).

    An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed.

A simple example

6

- Type 2+2 after the prompt > and hit enter
- 2+2 is an ***expression***
- Type Yikes<-2+2 after the prompt > and hit enter
- Here we are ***assigning*** of a value to a name
- Type Yikes at the prompt and what do we expect see?

A simple Example

7

If we type:
2+2
Yikes<-2+2
Yikes
We get:
> 2+2
[1] 4
> Yikes<-2+2
> Yikes
[1] 4

Case Sensitive

8

- "YIKES" and "yikes" are not the same to R as it is case sensitive
- Watch out for this!

```
> YIKES
Error: object 'YIKES' not found
> yikes
[1] 4
```

Vectors and assignment

9

- The entities that R creates and manipulates are known as objects.
- These may be variables or vectors or matrices or data sets or results or lists
- During each R session the objects we create are stored by name
- We can display the objects currently stored in R:

```
> objects()
[1] "yikes"
```

Assignment and data creation

10

`<-`	assign
`c(…)`	combine arguments into a vector
`seq(x)`	generate sequence from 1 to *x*
`seq(from,to,by)`	generate sequence with increment *by*
`from:to`	generate sequence *from* .. *to*
`rep(x,times)`	replicate *x*

```
> x <- 1
> y <- "A"
> my.vec <- c(1, 5, 6, 10)
> my.nums <- 12:24
> x
[1] 1
> y
[1] "A"
> my.vec
[1]  1  5  6 10
> my.nums
 [1] 12 13 14 15 16 17 18 19 20 21 22 23 24
```

Data Structures

11

R operates on named ***data structures***.

- The simplest structure is the numeric ***vector***, which is a single entity consisting of an ordered collection of numbers
- ***matrices* or more generally *arrays*** are multi-dimensional generalisations of vectors.
- *lists* are a general form of vector in which the various elements need not be of the same type, and are often themselves vectors or lists.
- ***data frames*** are matrix-like structures, in which the columns can be of different types. Think of data frames as 'data matrices' with one row per observation but with (possibly) both numerical and categorical variables. Think back to the data sets in STAT1008 and STAT1003!

Examples to come!

Vectors

12

- **Information on a vector**

`length(x)`	number of elements
`names(x)`	get or set names

- **Indexing (*number, character (name), or logical*)**

`x[n]`	*n*th element
`x[-n]`	all but the *n*th element
`x[a:b]`	elements *a* to *b*
`x[-(a:b)]`	all but elements *a* to *b*
`x[c(…)]`	specific elements
`x["name"]`	"*name*" element
`x[x > a]`	all elements greater than *a*
`x[x %in% c(…)]`	all elements in the set

Vectors: Example

13

- To assign a vector in R, we may issue the command:

```
>X <- c(1,2,5,6,9)
```

- There is no reason why the arguments must be numerical; they may be previously defined scalars, or even previously defined vectors:

```
>Y <-c(3,X,10,X,2)
> Y
[1]  3  1  2  5  6  9 10  1  2  5  6  9  2
```

Vectors: Subscripting

14

A vector (or matrix) name followed by square brackets, indicates an element of that vector (or matrix).

```
> X <- c(1,2,5,6,9)
> Y <-c(3,X,10,X,2)
> Y
[1]  3  1  2  5  6  9 10  1  2  5  6  9  2
> Y[2]
[1] 1
> Z<-Y[3:7]
> Z
[1]  2  5  6  9 10
```

Vectors: Subscripting

15

```
> Z<-Y[X]
> Z
[1] 3 1 6 9 2
> Z[3]<-7
> Z
[1] 3 1 7 9 2
> Z[Y[2:6]]<-c(11,12,13,14,15)
> Y[2:6]
[1] 1 2 5 6 9
> Z
[1] 11 12  7  9 13 14 NA NA 15
```

Arrays and Matrices

16

- There are several ways to create matrices in R. First, if we have previously defined vectors, we can use them to make a matrix:

```
> X <- c(1,2,3)
> Y <-c(4,5,6)
> Z <-c(7,8,9)
> M <- cbind(X,Y,Z)
> M
     X Y Z
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
```

- The function 'cbind()' creates a matrix with columns given by the vectors provided as arguments and in the order provided.

Arrays and Matrices

17

- rbind()

```
> M <-rbind(X,Y,Z)
> M
  [,1] [,2] [,3]
X    1    2    3
Y    4    5    6
Z    7    8    9
```

Arrays and Matrices

18

```
> X <- c(1,2,3,4,5,6,7,8,9)
> M<-matrix(X,ncol=3)
> M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> M <-matrix(X,ncol=3,byrow=T)
> M
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

We can create a matrix from a single vector using the 'matrix()' function, which takes optional arguments.

The optional argument 'byrow=T' tells R that the matrix to be created should be filled out by rows, instead of by columns which is the default method.

Lists

19

- A list is a list of other data structures, each named and accessible by that name.
- We will use lists with the function 'lsfit()' which does least squares regressions.
- lsfit() returns a list whose contained structures include such things as:
  - a vector of the parameter estimates
  - a vector of the residuals
- To access a member of a list, we use the '$' operator

Lists and lsfit()

20

```
> Y <- c(10,20,30,41)
> X <- c(1,2,3,4)
> reg.y.on.x <- lsfit(X,Y)
> reg.y.on.x$coef
Intercept     X
   -0.5    10.3
> reg.y.on.x$resid
[1]  0.2 -0.1 -0.4  0.3
>
```

To run a simple linear regression of 'Y' on 'X' (NOTE: As we will see, to run a multiple linear regression, we will use the same commands structure, but 'X' will be a matrix, and not simply a vector)

Naming in R

21

- Mostly free form but here are some rules and tips:
  - don't use the variable names 'c' 'q', 'T', 'F', 't', try() since they all mean something in R.
  - don't use arithmetic symbols
  - R name should begin with a letter
  - Don't use the assignment function "<-"
  - It is useful to give your structures names which indicate what they are (though you should remember that you may have to type the name of your variable many times)

Objects

22

- To view a listing of your current structures, use the function 'ls()';
- To tell R that you no longer want it to keep a particular structure, use the function 'rm()' and include the name of the structure you want removed between the parentheses.

Dataframes

23

- Most of the modelling commands in R such as 'lm()', which performs linear regression modelling take dataframes as input.
- Dataframes have most of the attributes of a matrix (their elements can be accessed using the square bracket notation used for matrices)
- But they also have the advantages of lists!
  - We can name and access their components using the '$' operator
- We will use dataframes when reading in external data, usually in the form of csv files

Setting your working directory

24

- Before reviewing R commands, it is important for you to set up a working directory for your R work. (Particularly if you plan to use R in the public computer labs.) In order to set up R, we use the setwd() command as follows:

```
setwd("<path>")
```

- note / instead of \ in windows
- Or we can set the path using the following:
- In R go to ***File*** then ***Change dir...*** Then locate the relevant directory

Reading in data

25

- We can create dataframes using the 'read.csv()' command as follows:

```
myfirstdf<-read.csv("filename.csv",header=T)
```

- The option 'header=T' tells R to read and save the column and/or row names within the dataframe object.

Reading in data

26

- Suppose 'filename.csv' contained three columns named 'pred1', 'pred2' and 'resp'.
- We could access these columns individually as 'filename.csv$pred1','filename.csv$pred2', and 'filename.csv$resp'. Alternatively, we can use the function:

```
>attach(filename.csv)
```

- to allow us to access 'pred1', 'pred2' and 'resp' without needing to type the 'file.df$' prefix all the time.
- This will become extremely useful when we start to fit complicated models to data.

Functions- Intrinsic

27

- **Arithmetic Functions**
  - Numerical functions :

    +, -, *, /, ^, <-, :, log(), sin(), cosh(), mean(), var()

  - Matrix functions

    t(), cbind(), solve(), %*%

- The numeric functions work on vectors as one would expect – adding subtracting, logging, etc., on an *element-by-element* basis
- For the binary operators (+, -, *, /), if the two vectors to be operated on have different lengths, the shorter one is repeated as many times as necessary to complete the calculation

Example

28

```
> X <- c(1,2)
> Y <-rep(1,12)
> X+Y
[1] 2 3 2 3 2 3 2 3 2 3
  2 3
```

- The function 'rep(X,n)' creates a vector which contains 'n' replications of the vector 'X'.
- Recall that if the two vectors to be operated on have different lengths, the shorter one is repeated as many times as necessary to complete the calculation

Functions- Intrinsic

29

```
> M1<-matrix(c(1,2,3,4),ncol=2,byrow=T)
> M1
    [,1] [,2]
[1,]   1    2
[2,]   3    4
> M2<-M1%*%M1
> M2
    [,1] [,2]
[1,]   7   10
[2,]  15   22
```

- Matrix Multiplication
- In order to multiply two matrices 'M1' and 'M2' in the usual sense of matrix multiplication, we use the binary operator '%*%'

Functions - Intrinsic

30

```
> solve(M1)
    [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

- to invert a matrix, we use the function 'solve()'
- for a 2x2 Matrix the Inverse is:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}^{-1} = 1/DET * \begin{vmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{vmatrix}$$

with DET $= a_{11}a_{22}-a_{12}a_{21}$

Functions - Intrinsic

31

- Flow of control functions - if(), ifelse(), for(), while(), q()
- Statistical Functions - lsfit(), lm(), glm()
- Graphical Functions - plot(), lines()
- Examples to come (tutorials and worksheets)

Functions - Intrinsic

32

- There are actually two types of graphical functions, those which end a previous plot and start a new one, and those which merely add to the current plot being created.
- Examples in worksheets and tutorials

Script Files

33

- We have been typing directly into the R command line.
- What if we could save commands in an R source file to be run anytime?
- By default, launching **R** starts an interactive session with input from the keyboard and output to the screen.
- We can get input come from a script file (a file containing **R** commands) and direct output to a variety of destinations.

Script Files

34

- Aconvenient method for using R.
- You can create your commands in a text editor and then source them to R
- If there are errors in your commands you can simply and easily edit your commands in the text editor and then re-source them to R

Script Files

35

- **Input**
- The **source( )** function runs a script in the current session. If the filename does not include a path, the file is taken from the current working directory.
- # input a script source("myfile.txt")

myfile.txt

36

- "myfile.txt" file available on wattle for you to try running yourself

```
# Our first script :o)
y=c(12,15,28,17,18)
x=c(22,39,50,25,18)
mean(y)
mean(x)
plot(x,y)
```

Script Files

37

- **Output**
- The **sink( )** function defines the direction of the output.
- The **append** option controls whether output overwrites or adds to a file. The **split** option determines if output is also sent to the screen as well as the output file.

```
#first run script
>source("myfile.txt")

# then direct output to a file
>sink("myoutputfile.out", append=FALSE, split=TRUE)

#print output to "myoutputfile.out" as well as to screen
> x
[1] 22 39 50 25 18
> y
[1] 12 15 28 17 18
> mean(x)
[1] 30.8

# close the output file
>sink()
```

User Defined Functions

38

```
fun.name <- function(args) {
  statements
  x or return(x)
}
```

- arguments(args) passed by value
- result of last statement is return value

Function Example

39

```
> square<-function(x){
+ y=x^2
+ y
+ }
> square(2)
[1] 4
```

- The '+' prompt indicates that we have an unfinished R line (which R can determine due to our unclosed curly brackets)
- The variables 'x' and 'y' inside our functions are just dummy variables, and bear no relation to any variables 'x' and 'y' which we might have previously created

Function Example

40

- What happens if we pass a vector?

```
> x<-c(3,5)
> square(x)
[1]  9  25
>
```

Function Example

41

- What happens if we just type "square" without any arguments?

```
> square
function(x){
y=x^2
y
}
```

Help

42

```
> help(mean)
> ?median
> help("[")
> example(mean)
> help.search("regression")
> RSiteSearch("genetics")
> http://www.r-project.org/
```

(try these yourself!!!)

R Workspace

43

```
ls()                list objects in workspace
rm(…)               remove objects from workspace
rm(list = ls())     remove all objects from workspace
save.image()        saves workspace

#                   comments
```

Example (at end of section 1)

44

- The investment in certain share portfolios (*x*) and the value after a year (*y*)in $000 are given in the table below.

x	1	2	4	5	8
y	3	3	7	6	12

- Fit a regression line using R

Regression Example

45

```
> X<-c(1,2,4,5,8)
> Y<-c(3,3,7,6,12)
> plot(X,Y)
> reg.out<-lsfit(X,Y)
> reg.out$coef
Intercept     X
     1.0      1.3
>abline(lsfit(X,Y)$coeff)
```