R for Newbie

Instructor: Ehsan Karim
Writeup: Justin Harrington
Department of Statistics, UBC

September 23, 2008

Overview:

- Introductions
- Objects
- Importing Data
- Statistical Functions
- Plotting
- Functions
- Logical Constructs & Looping
- Further development

Introductions | Goals

- To give you, the user, an introduction to R and what it can do

- To enable you to continue your own education by understanding the fundamentals, and what resources are available

Introductions - R

What is it?

"R is an integrated suite of software facilities for data manipulation, calculation and graphical display. It includes

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either on-screen or on hardcopy, and
- a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities."*

*From the R-Project website

What are the alternatives, and when do you use them?

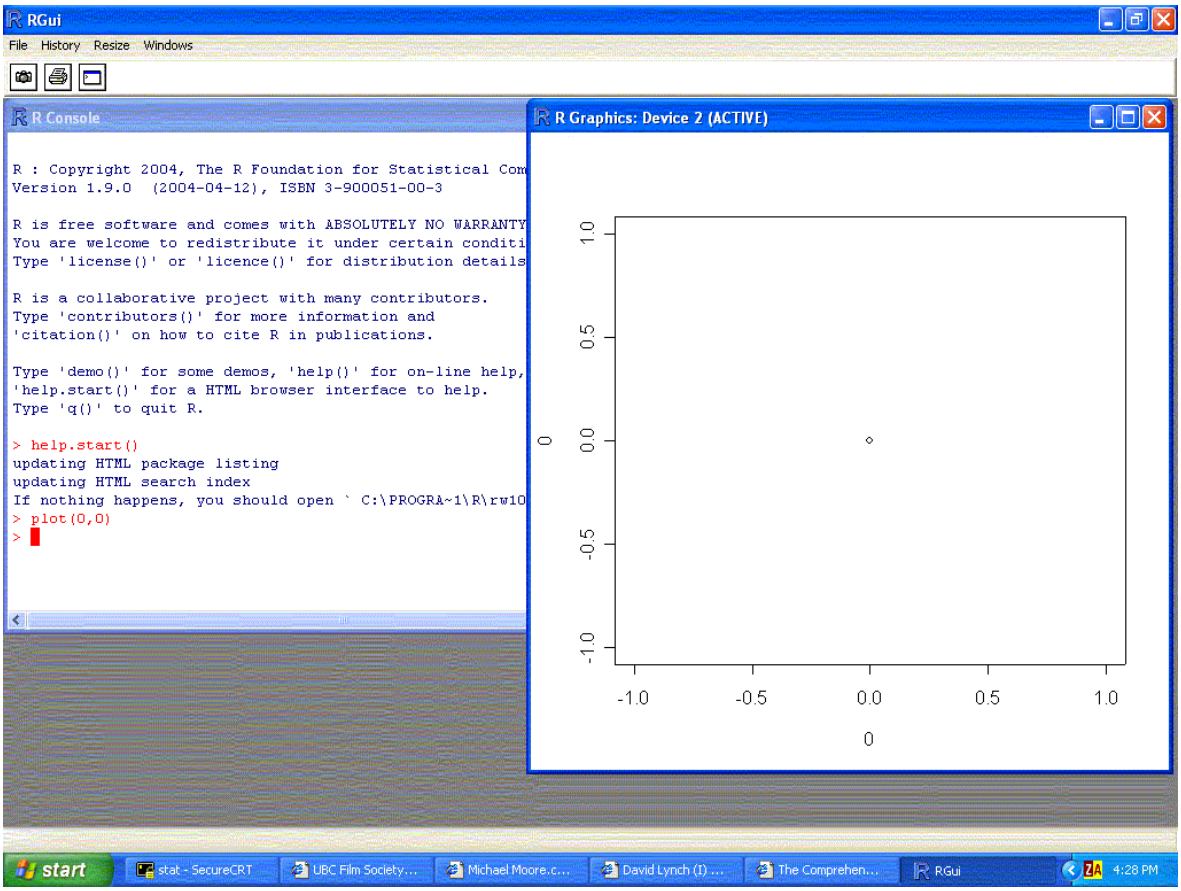
- SAS - Is second to none in terms of handling enormous datasets, and performing "vanilla" statistical procedures like regression, ANOVA, tabulating etc. Doesn't handle anything out of the ordinary.

- JMP-IN - A very intuitive tool that has a reasonable library of statistical tests that can be performed by point-and-click operations. However, it's quite slow and cannot easily be expanded beyond what is built in.

- Microsoft Excel - Excel is loaded on almost every computer in the world (much to Bill Gates delight), and is an interface that most people know how to use. There are a few simple tests available like ANOVA and regression.*

*These are not loaded by default. To get access, see the reference manual.

The R environment



The R environment

- Any commands are typed in the Console window
- Us the up-arrow to retrieve previous commands
- To interrupt a process, try the ESC key (in Windows)
- The plot window automatically appears when the first plot command is executed
- To leave R, type `q()` and press enter. If you wish to save your working directory, then answer yes to the prompt.

Exercise:

1. Open R

2. Type the following in the Console window:

   ```
   x <- "Hello World"
   x
   ```

   and observe what happens.

3. Type `q()` and answer yes to the prompt.

4. Open R again, and type `x` in the console window

Objects

- What is an object
  - their creation,
  - naming conventions
- Vectors
- Matrices
- Recycling
- Dataframes

What is an object?

- The jar analogy:
  - R stores information in objects which are saved in your working directory.
  - You can think of an object as a jar that contains information, and has a name on the front.

- Valid names include any letter of the alphabet, numbers (as long as the name does not start with a number). You can use decimal points (though I don't recommend it), you cannot use any special characters (`_ / *` etc), and the names are case sensitive. Avoid predefined names.

- To create an object, you assign something to it using `<-`. E.g. in the previous exercise, you created an object `x`.

- If you refer to an object, it is as if you are inserting the objects contents into the command.

Exercise: Type the following commands, and observe the output:

```
x <- 5
x
y <- 1
y
x + y
x * y
x / y
```

You should see the following:

```
> x <- 5
> x
[1] 5
> y <- 1
> y
[1] 1
> x + y
[1] 6
> x * y
[1] 5
> x / y
[1] 5
```

Objects *cont.*

- In the previous exercise, we have used some mathematical operators. Some more include:

`abs(x)`	absolute value	`sqrt(x)`	square root
`sin(x)`	sine	`cos(x)`	cosine
`log(x)`	Natural log	`exp(x)`	e^x
`x^y`	x^y		

- To see what objects are currently in your working directory, use the command `ls()`. To remove an object, the command is `rm(`*name*`)`. E.g.

```
> ls()
[1] "x" "y"
> rm(x)
> ls()
[1] "y"
```

Exercise: Calculate 5^{2+1} directly, and then by creating the objects $x = 5, y = 2, z = 1$ and calculating x^{y+z}.

```
  > 5^(2+1)
[1] 125
> x <- 5; y <- 2; z <- 1
# note use of semi-colons
> x^(y+z)
[1] 125
```

Vectors

- A vector is the mathematical name for a column of data, like

$$\begin{pmatrix} 5 \\ 2 \\ 4 \\ 6 \end{pmatrix}$$

- The command in R to create an object containing this vector is `MyVect <- c(5,2,4,6)`

- You can also create a vector using this method from other vector/scalar objects. For example, if you have `x<-c(5, 2)` and `y <- c(4,6)` then `MyVect <- c(x,y)` would have the same result.

- The other method of creating a vector is `rep`, which is used when you want a vector of a certain length to contain all the same values. For example, `MyVect <- rep(0, 20)` would create a vector of length 20, and all the elements being 0.

Vectors *cont.*

- Sequences of numbers (i.e. 1, 2, 3, ...) appear quite frequently in R. There are two methods of generating these: *Method One:* `1:5` produces

```
> 1:5
 [1]  1  2  3  4  5
```

Method Two: `seq(1,5)` produces

```
> seq(1,5)
 [1]  1  2  3  4  5
```

- The advantage of the second method is that there's a lot more that the `seq` command can do.
  - `seq(`*from, to, length*`)` e.g. `seq(1,10, length=4)` gives

```
> seq(1,10, length=4)
[1]  1  4  7 10
```

  - `seq(`*from, to, by*`)` e.g. `seq(1,10, by=2)` gives

```
> seq(1,10, by=2)
[1] 1 3 5 7 9
```

Vectors *cont.*

- To select elements from a vector, use square brackets. e.g. `MyVect[ i ]` selects the i'th element from the vector `MyVect`.

- Examples:

```
> MyVect <- 10:1
> MyVect
 [1] 10  9  8  7  6  5  4  3  2  1
> MyVect[3] # Selects third element
[1] 8
> MyVect[c(3,5,1)]
# selects third, fifth and first elements
[1]  8  6 10
> MyVect[3:7]
# selects third through seventh elements
[1] 8 7 6 5 4
> x <- c(1,3,5)
> MyVect[x]
# selects the elements specified in x
[1] 10  8  6
```

- Note: whenever R sees an object, it simply replaces the object by its values. So, `MyVect[x]` is exactly the same as `MyVect[c(1,3,5)]`

Exercise:

1. Create an object called `EvenVect` containing a sequence of even numbers between 2 and 100.

2. Select only the odd elements of the vector i.e. the first, third, fifth, ... elements containing $2, 6, 10, \ldots$.

There are many ways of doing this - two are
Approach One:

```
> EvenVect <- seq(2, 100, by=2)
> EvenVect
 [1]   2   4   6   8  10  12  14  16  18  20
      22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58
      60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96
      98 100
> EvenVect[seq(1, 50, by=2)]
 [1]  2  6 10 14 18 22 26 30 34 38 42 46 50
     54 58 62 66 70 74 78 82 86 90 94 98
```

Approach Two:

```
> EvenVect <- (1:50)*2
> EvenVect
 [1]   2   4   6   8  10  12  14  16  18  20
      22  24  26  28  30  32  34  36  38
[20]  40  42  44  46  48  50  52  54  56  58
      60  62  64  66  68  70  72  74  76
[39]  78  80  82  84  86  88  90  92  94  96
      98 100
> EvenVect[(1:25)*2-1]
 [1]  2  6 10 14 18 22 26 30 34 38 42 46 50
     54 58 62 66 70 74 78 82 86 90 94 98
```

Matrices

- A matrix is the mathematical name for an "array" of data like

$$\begin{pmatrix} 2 & 3 & 4 \\ 1 & 9 & -22 \end{pmatrix}$$

and their behaviour is very similar to vectors.

- To create a matrix, you give the command `matrix` a vector, and tell it how many rows and/or columns there are. By default it fills the matrix by column (to fill by row, set the argument `byrow=T`).

- To create the above matrix, the command would be `MyMatrix <- matrix(c(2, 1, 3, 9, 4, -22), nrow=2, ncol=3)` . Alternatively, to create a matrix with all zeros, the command would be `MyMatrix <- matrix(0, nrow=2, ncol=3)`.

Matrices *cont.*

- Note that the following is equivalent:

```
MyMatrix <- matrix(c(2, 1, 3, 9, 4, -22),
nrow=2, ncol=3)
MyMatrix <- matrix(c(2, 1, 3, 9, 4, -22),
ncol=3)
MyMatrix <- matrix(c(2, 1, 3, 9, 4, -22),
nrow=2)
```

- An alternative method of creating matrices is by "binding" together two or more vectors/matrices together using the commands `rbind` (row bind) and `cbind` (column bind). E.g.

$$\texttt{cbind}\left(\begin{pmatrix}1\\4\\7\end{pmatrix},\begin{pmatrix}2&3\\5&6\\8&9\end{pmatrix}\right) = \begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix} \qquad \texttt{rbind}\left(\begin{pmatrix}1&2&3\end{pmatrix},\begin{pmatrix}4&5&6\\7&8&9\end{pmatrix}\right) = \begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}$$

Exercise: Create a matrix, called `MyMatrix`, by coding the previous example using:

1. `cbind`;
2. `rbind`; and
3. `matrix`

Answers:

1.
```
MyMatrix<-cbind(c(1,4,7),
matrix(c(2,3,5,6,8,9), ncol=2,byrow=T))
```

2.
```
MyMatrix <- rbind(1:3, matrix(4:9, nrow=2,
byrow=T))
```

3.
```
MyMatrix <- matrix(1:9, nrow=3, byrow=T)
```

Matrices *cont.*

- To refer to elements within a matrix, a similar syntax to vectors is used, except this time there are two "co-ordinates" [*row*,*column*] e.g.

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

- There is also a convenient way of selecting entire rows and columns.
  - `x[i,]` selects row i
  - `x[,j]` selects column j

  For example,

```
> MyMatrix[2,]
[1] 4 5 6
> MyMatrix[,3]
[1] 3 6 9
```

Matrices *cont.*

- You can refer to row and column names where they exist.

```
> x <- 1:3; y <- 4:6; z <- cbind(x,y)
> z
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
> z[,"x"]
[1] 1 2 3
```

- More generally, when data is imported from a file with headings, using labels rather than column numbers is very convenient.

- The command `dim` returns its dimensions of a matrix. For example `dim(z)` returns

```
> dim(z)
[1] 3 2 # Number of rows, Number of columns
```

Recycling

- In the previous examples, dimensions were chosen that were deliberately consistent with one-another.

- However, if the dimensions are not consistent, then R can do seemingly unpredictable things. For example,

```
> x <- 1:3; y <- 1:2; cbind(x,y)
     x y
[1,] 1 1
[2,] 2 2
[3,] 3 1
Warning message:
number of rows of result
is not a multiple of vector
length (arg 2) in: cbind(x, y)
```

  This is not a good thing!

Recycling *cont.*

- However, suppose you wanted to create the following matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 2 & 2 \\ 3 & 2 & 1 \end{pmatrix}$$

There are two ways of doing this:

Method One:

```
> cbind(1:3, rep(2, 3), 3:1)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    2    2
[3,]    3    2    1
```

Method Two:

```
> cbind(1:3, 2, 3:1)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    2    2
[3,]    3    2    1
```

- Notice that in the second case, R actually repeats the 2 enough times to make it match the dimensions of the other two vectors.

- This is called "Recycling" and is a very powerful tool for speeding up processes.

Exercise: Create the following matrices:

1. $\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$

2. $\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 2 & 3 \\ 2 & 2 & 2 & 0 & 1 & 2 & 3 \\ 3 & 3 & 3 & 0 & 1 & 2 & 3 \end{pmatrix}$

Hint: use recycling and sequences.

Answers:

```
matrix(1:3, nrow=3, ncol=3)
cbind(matrix(1:3, nrow=3, ncol=3), 0,
matrix(1:3,nrow=3, ncol=3, byrow=T))
```

Dataframes

- A dataframe is very much like a matrix, except it allows different columns to have different type (i.e. numbers and text).

- E.g.

```
> x <- 1:5
> y <- letters[1:5]
> z1 <- cbind(x,y)
> z1
     x   y
[1,] "1" "a"
[2,] "2" "b"
[3,] "3" "c"
[4,] "4" "d"
[5,] "5" "e"
```

```
> z2 <- data.frame(x,y)
> z2
  x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
```

- Note in the situation where a matrix was created, the numbers have been turned into text, whereas with the dataframe it appears that both types are accommodated in the same object.

Importing Data

- An introduction to `read.table`
  - CSV, Tab Delimited and Fixed Formats
  - Todays datasets
  - Missing values
- Factors

Intro to `read.table`

- Until now all our data has been typed by hand. In the real world, however, most of our data will already be loaded in a file (e.g. text file, Excel, etc.), and we'll wish to load it into R.

- The command most commonly used for this is `read.table`. It's syntax is

```
read.table(file, header = FALSE, sep = "",
 quote = "\"'", dec = ".", row.names,
 col.names, as.is = FALSE, na.strings = "NA",
 colClasses = NA, nrows = -1, skip = 0,
 check.names = TRUE, fill = !blank.lines.skip,
 strip.white = FALSE, blank.lines.skip = TRUE,
 comment.char = "#")
```

  As you can see, there are many options!

- Before we get carried away with this command, let's first look at the different types of file most commonly encountered.

CSV, Tab Delimited and Fixed Format Files

- Comma Separated Variable (CSV)

```
Difference,EPP,Mono/Poly
-100,0.08,M
-55,0.08,M
-35,0.14,M
```

- Tab Delimited

```
Number  Length
1.010050167     8.166169913
1.23367806      6.685894442
1.010050167     6.553504862
1.447734615     6.553504862
```

- Fixed Format

```
Sex          GPA   Seat  Alchol
Female     2.600    M       15
Male       2.700    M       14
Female     3.000    F        *
Female     3.110    F       10
```

Todays Datasets

- `Monogamy.csv` - Derived from Figure 2a of: Dixon, A., Ross, D., O'Malley, S.L.C., & Burke, T. (1994) "Paternal investment inversely related to degree of extra-pair paternity in the reed bunting", *Nature* **371**, 698 - 700.

- `StayingPower.tab` - Derived from Figure 4 of: Gomendio, M. & Roldan, E.R.S (1991) "Sperm competition influences sperm size in mammals", *Proc. R. Soc. Lond. B* **243**, 181-185.

- `ucdavis2.txt` - Utts, J.M. & , Robert F. Heckard (2003) "Mind on Statistics", Duxbury Press.

Back to `read.table`

- There are only three commonly used parts of `read.table`:
  - `file`: the path and filename of the dataset (the path used is relative to the current working directory)
  - `header`: If you set this to TRUE, then the first row is consider the column names; if it is left as false, then it incorporates the first row into the dataset
  - `sep`: for a csv file, set this to `","`; for a tab-delimited file set this to `"\t"`, and for a fixed format file set this to `""`.

- Don't forget to assign the output of a `read.table` to an object!

- (Hypothetical) examples:

```
monogamy <- read.table(
"z:/RCourse/Monogamy.csv", sep=",", header=T)
monogamy <- read.table(
"z:/RCourse/Monogamy.tab", sep="\t", header=T)
monogamy <- read.table(
"z:/RCourse/Monogamy.txt", sep="", header=T)
```

Missing Values

- It is very rare that a dataset has no missing values, and as a result it is necessary to deal with these occurrences when the data is imported.

- In R, a missing value is coded `"NA"`. However, in your dataset it might be coded as something different, and it is necessary to tell `read.table` what that code for a missing value is.

- To deal with this, simply add in `na.strings="NA Identifier"`

Exercise:

1. Look at each of the datasets in a text viewer. Note in particular
   (a) is there a header?
   (b) how are the columns separated (i.e. CSV, tab etc.)
   (c) how are missing values coded?

2. Import the datasets using `read.table`. The following table gives the file formats, and the objects you should save the data under.

Filename	Object Name	File Type
Monogamy.csv	`monogamy`	CSV
StayingPower.tab	`blush`	Tab-Delimited
ucdavis2.txt	`utts`	Fixed Width

   Note that there are missing values in the ucdavis2.txt table, and they are coded as "*".

Answers:

```
monogamy <- read.table("z:/RCourse/Monogamy.csv", sep=",", header=T)
blush <- read.table("z:/RCourse/StayingPower.tab", sep="\t", header=T)
utts <- read.table("z:/RCourse/ucdavis2.txt", sep="", header=T,
      na.strings="*")
```

Factors

- A factor is the statistical expression for something that can only take on certain, defined levels.
- A very good example of a factor is given in the monogamy dataset, where the Mono.Poly column only takes on "M" or "P".
- Normally R will automatically recognize these as factors, and incorporate that into the structure of the dataframe.
- Commands associated with factors include:
  - `is.factor` -

    ```
    > is.factor(monogamy[,"Mono.Poly"])
    [1] TRUE
    ```

  - `as.factor` -

    ```
    monogamy[,"Mono.Poly"] <-
    as.factor(monogamy[,"Mono.Poly"])
    ```

  - `levels` -

    ```
    > levels(monogamy[,"Mono.Poly"])
    [1] "M" "P"
    ```

Exercise Confirm that the column "Seat" in the dataset utts is a factor, and find its levels.

```
> is.factor(utts[,"Seat"])
[1] TRUE
> levels(utts[,"Seat"])
[1] "B" "F" "M"
```

Statistical Functions

- Random Number Generation
- Summary Statistics
- Regression and ANOVA

Random Number Generation

- Uniform Distribution:
  `runif(`*n, min, max*`)` – generates *n* observations from a uniform distribution between *min* and *max*. Defaults are `min=0, max=1`.

```
> runif(5, 10, 12)
[1] 11.28290 10.50572 10.19792 10.69047 11.73497
```

- Normal distribution:
  `rnorm(`*n, mean, sd*`)` – generates *n* observations from a normal distribution with mean *mean* and standard deviation *sd*. Defaults are `mean=0, sd=1`.

```
> rnorm(5, 1, 5)
[1]  9.918982 -2.796923 -3.718555  1.053755 -1.118268
```

- Other distributions include:

Poisson - `rpois`	Beta - `rbeta`	Exponential - `rexp`
Binomial -`rbinom`	Gamma -`rgamma`	Chi-squared - `rchisq`

Summary Statistics

- `mean(`*x*`)` - calculates the mean of *x*
- `var(`*x*`)` - calculates the variance (the standard deviation squared) of *x*
- `median(`*x*`)` - calculates the median of *x*
- `summary(`*x*`)` - calculates various summary statistics of *x*
- `cor(`*x*`,` *y*`)` - calculates the correlation between *x* and *y*
- `sum(`*x*`)` - sums up all the elements in *x*

```
> x <- rnorm(100)
> mean(x)
[1] -0.05161488
> var(x)
[1] 1.329073
> sum(x)
[1] -5.161488
```

```
> median(x)
[1] -0.09966405
> y <- x + rnorm(100)
> cor(x,y)
[1] 0.773494
```

```
> summary(x)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.50900 -0.85380 -0.09966 -0.05161  0.75670  3.13200
```

Exercise:

1. With the object `blush`, calculate

   (a) the correlation between number and length

   (b) the correlation between log(number) and log(length)

2. With the object `monogamy` calculate the summary statistics

```
> cor(blush[,"Number"], blush[,"Length"])
[1] 0.6417976
> cor(log(blush[,"Number"]), log(blush[,"Length"]))
[1] 0.573943

> summary(monogamy)
   Difference              EPP             Mono.Poly
 Min.   :-100.00000   Min.   :-0.210000   M:8
 1st Qu.: -35.00000   1st Qu.:-0.080000   P:5
 Median :  23.00000   Median :-0.050000
 Mean   :  -0.07692   Mean   :-0.007692
 3rd Qu.:  25.00000   3rd Qu.: 0.080000
 Max.   : 100.00000   Max.   : 0.150000
```

Regression and ANOVA

- Review of regression and ANOVA
- The basic syntax of linear regression is:

  `lm(`*model*`, data=`*dataset*`)`

- The model is expressed in the form *Response* ~ *Explanatories*.
  E.g. with the blush dataset if the response is Number and the explanatory Length, then

```
> lm(Number ~ Length, data=blush)

Call:
lm(formula = Number ~ Length, data = blush)

Coefficients:
(Intercept)       Length
    -0.3101       0.2571
```

- More common practise would be to assign the regression output to an object, and then use the command `summary` to display more diagnostics.

Regression and ANOVA *cont.*

```
> BlushReg <- lm(Number ~ Length, data=blush)
> summary(BlushReg)

Call:
lm(formula = Number ~ Length, data = blush)

Residuals:
      Min         1Q    Median         3Q        Max
-0.779629 -0.182134  0.003305  0.086763  0.725044

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.31005    0.62485  -0.496  0.62745
Length       0.25713    0.08211   3.131  0.00736 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3679 on 14 degrees of freedom
Multiple R-Squared: 0.4119,     Adjusted R-squared: 0.3699
F-statistic: 9.806 on 1 and 14 DF,  p-value: 0.007359
```

Regression and ANOVA *cont.*

- Note that if our model had two explanatories, then the model would be entered as *Response* ~ *Exp1* + *Exp2*.

- There are two methods of doing ANOVA. The first method involves creating a linear regression object using `lm`. and then using the command `anova`. A second command, `aov`, does both steps at once.

  ### Method One

```
> MyReg <- lm(Alchol ~ Seat, data = utts)
> anova(MyReg)
Analysis of Variance Table

Response: Alchol
           Df Sum Sq Mean Sq F value    Pr(>F)
Seat        2 1120.8   560.4  13.167 4.171e-06 ***
Residuals 205 8725.2    42.6
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1
```

Method Two

```
> MyAov <- aov(Alchol ~ Seat, data = utts)
> summary(MyAov)
             Df Sum Sq Mean Sq F value    Pr(>F)
Seat          2 1120.8   560.4  13.167 4.171e-06 ***
Residuals   205 8725.2    42.6
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1
```

Exercise: With the dataset utts

1. Build a model with Ideal Height (`IdealHt`) as the response, and Height as the explanatory. Is it significant?

2. Does drinking alcohol have an effect on your GPA?

With the dataset blush

1. Fit a regression line between length and number

2. Fit a regression line between log(length) and log(number)

```
> summary(lm(IdealHt~Height, data=utts))

Call:
lm(formula = IdealHt ~ Height, data = utts)

Residuals:
    Min      1Q  Median      3Q     Max
-6.0356 -1.1269 -0.1225  1.2339  5.4165

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 14.27119    2.15413   6.625 2.45e-10 ***
Height       0.81738    0.03225  25.345  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.883 on 229 degrees of freedom
Multiple R-Squared: 0.7372,     Adjusted R-squared: 0.7361
F-statistic: 642.4 on 1 and 229 DF,  p-value: < 2.2e-16
```

```
> summary(aov(GPA~Alchol, data=utts))
             Df Sum Sq Mean Sq F value   Pr(>F)
Alchol        1  2.091   2.091  7.2914 0.007522 **
Residuals   200 57.347   0.287
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1
```

```
> summary(lm(Number~Length, data=blush))

Call:
lm(formula = Number ~ Length, data = blush)

Residuals:
      Min         1Q    Median         3Q        Max
-0.779629 -0.182134  0.003305  0.086763  0.725044

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.31005    0.62485  -0.496  0.62745
Length       0.25713    0.08211   3.131  0.00736 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3679 on 14 degrees of freedom
Multiple R-Squared: 0.4119,     Adjusted R-squared: 0.3699
F-statistic: 9.806 on 1 and 14 DF,  p-value: 0.007359
```

```
> summary(lm(log(Number)~log(Length), data=blush))

Call:
lm(formula = log(Number) ~ log(Length), data = blush)

Residuals:
     Min       1Q   Median       3Q      Max
-0.53912 -0.08691  0.02316  0.07230  0.42887

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  -1.7308     0.8334  -2.077   0.0567 .
log(Length)   1.0857     0.4140   2.622   0.0201 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’
                0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.2343 on 14 degrees of freedom
Multiple R-Squared: 0.3294,     Adjusted R-squared: 0.2815
F-statistic: 6.877 on 1 and 14 DF,  p-value: 0.02008
```

An Introduction to Plotting

Plotting - the basics

- The best and worst thing about R

- The generic 2-dimensional plotting command in R is `plot`, and what it produces depends on the inputs given.

X	Y	Plot Type
Continuous	–	Index Plot
Continuous	Continuous	Scatter Plot
Factor	Continuous	Box-plot for each factor
Factor	Factor	Stacked Bar Chart

Plotting - the basics *cont.*

The syntax for plot is `plot(x, ...)` where `...` includes:

- `y` - the second vector of data (if required)
- `type=` - the type of plot e.g. point (`type="p"`) or line (`type="l"`)
- `xlab=` - the x-axis label (in quotes)
- `ylab=` - the y-axis label (in quotes)
- `main=` - the title (in quotes)
- `pch=` - the type of point to use (if `type="p"`); if it is text, then `pch="x"`, else a number (experiment to find the best one)

- `lty=` - the type of line to use (if `type="l"`); a number (experiment to find the best one)

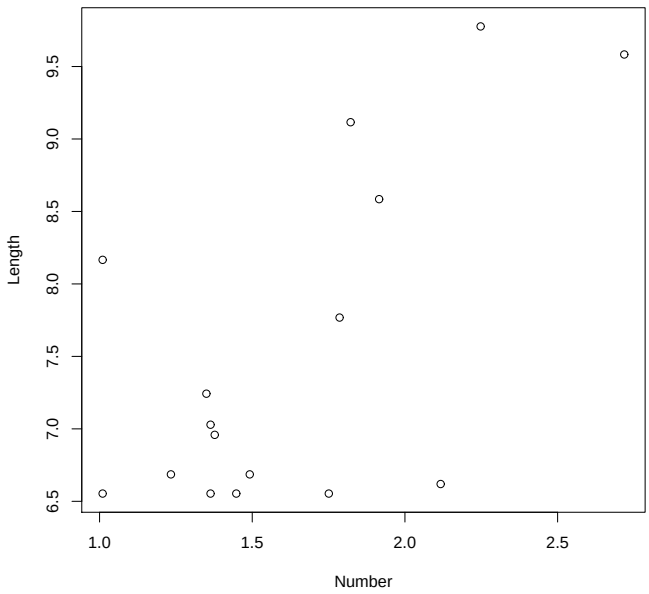
- `xlim= c(low, high)` - the limits of the x-axis (i.e. the highest and lowest number displayed)

- `ylim= c(low, high)` - the limits of the y-axis

Note that, with the exception of `x`, all the other arguments are optional.

Plotting - an example

`plot(blush[,"Number"], blush[,"Length"], type="p", xlab="Number", ylab="Length")+` produces a plot of Number (on the x-axis) versus Length (on the y-axis).



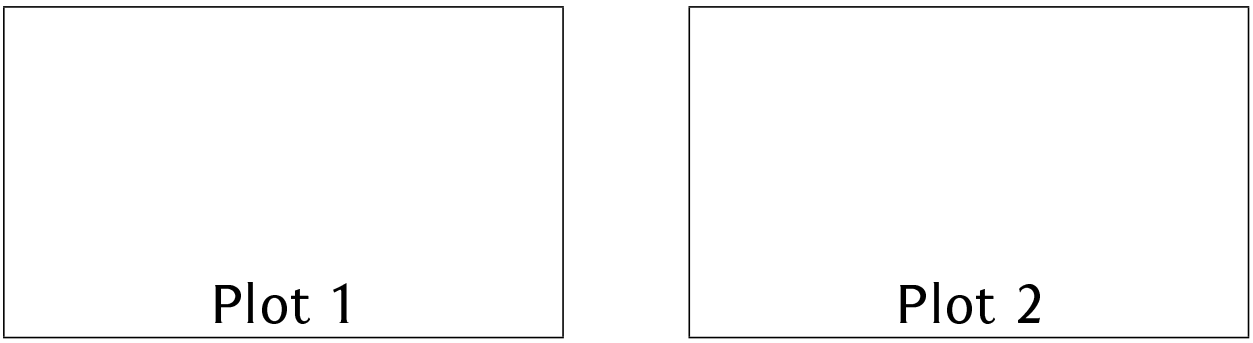
Note: `plot(blush)` produces exactly the same plot.

To put more than one plot on the same page, use the commands

```
par(mfrow=c(n,m))
par(mfcol=c(n,m))
```

before the plotting commands are given. These commands subdivide the plotting area into `n` rows by `m` columns, with the first command filling along the row first, and the latter along the columns first.

E.g. `par(mfrow=c(1,2))` would give



Adding to the current plot

- Once a plot has been created, it is possible to add further points/lines, with the commands `points` and `lines`.

- One thing to be careful of is that the existing plot's axes are large enough to contain all the points.

- For example (very typical):

```
x1 <- rnorm(10)
x2 <- rnorm(10)
y1 <- x1 + rnorm(10)
y2 <- x2 + rnorm(10)

plot(x1, y1, xlim=c(min(x1,x2), max(x1,x2)),
             ylim=c(min(y1, y2), max(y1,y2)),
             pch=1)
points(x2, y2, pch=3)
```

- What is being done?

Adding to the current plot *cont.*

- To add a line to the current plot, the `abline` command is used.

- There are many versions - the following is taken from the help file:

```
abline(a, b, untf = FALSE, ...)
abline(h=, untf = FALSE, ...)
abline(v=, untf = FALSE, ...)
abline(coef=, untf = FALSE, ...)
abline(reg=, untf = FALSE, ...)

Arguments
a,b the intercept and slope.
untf logical asking to untransform.
See Details.
h the y-value for a horizontal line.
v the x-value for a vertical line.
```

Other plots

- `boxplot` - if you give it a matrix, it will do a boxplot for each continuous column

- `hist` - produces a histogram of a vector

Exercise:

1. Produce a scatter plot of Difference vs EPP from the dataset monogamy, correctly labeling the axes, and giving it a title.

2. Produce a histogram of the GPA's in the dataset utts.correctly labeling the x-axis, and giving it a title.

3. Produce a boxplot of Alcohol Consumption by Seat from the dataset utts.

4. Plot a scatter plot of Number vs Length from the dataset blush. Overlay this with the regression line. (Hint - save the regression output into an object, and then use the command abline).

Answers:

```
plot(monogamy[,"Difference"], monogamy[,"EPP"],
     xlab="Difference",
     ylab="EPP", main="Plot of...")

hist(utts[,"GPA"], xlab="GPA", main="Plot of...")

plot(utts[,"Seat"], utts[,"Alchol"])

plot(blush)
x <- lm(Length~Number, data=blush)
abline(x, lty=3)
```

Functions

- Intro & syntax
- Arguments

Introduction to functions

- Functions are an extremely useful tool for developing your own code in a user friendly and organized way,

- In actual fact, you've already been using predefined functions already!

- Consider the following calculation for the skewness of a vector `MyVect`:

```
mean((x-mean(MyVect))^3)/
(sqrt(var(MyVect)))^3
```

-

-

- A better way is to define a function:

```
skewness<-function(x){
          y <- mean((x-mean(x))^3)/
          (sqrt(var(x)))^3
          return(y)
}
```

- To execute a function, type the name of the function, followed by ()'s, and provide any arguments as required. E.g. `skewness(MyVect)`.

- Note that the output from a function can be assigned to an object.

Intro to functions *cont.*

- The generic syntax of a function is as follows:
  name `<- function(`*arguments*`){`
  steps
  ...
  `return(...)`
  `}`

- In the previous example,
  - `x` was the argument
  - `y <- mean((x-mean(x))^3)/(sqrt(var(x)))^3` was the step
  - and `y` was returned.

Arguments

- The preferred method of getting data "into" the function.

- There are two ways of matching the arguments –

  – the same order as specified in the function definition

  – specify which argument you are giving data to, and then the ordering doesn't matter.

  For example,

```
AboutMe <- function(MyName, MyAge, HairColor){
    return(c(MyName, MyAge, HairColor))
}
> AboutMe("Ehsan", 78, "Blonde")
[1] "Ehsan" "18"     "Blonde"
> AboutMe(MyAge = 78, HairColor="Blonde",
MyName="Ehsan")
[1] "Ehsan" "78"     "Blonde"
```

- Revisiting the `seq` command.

Steps

- Any steps can go into a function.

- Caution should be used when using objects - if you refer to an object that already exists in the working directory, the results can be a little ambiguous.
  For example,

```
x <- 5
MyFunc1 <- function(x){
   print(x)
}
> MyFunc1(10)
[1] 10
```

- If an object is an argument in the function, then any references to that object take on the value in the function, and ignore the object in the working directory.

Steps *cont.*

- And just to make things more complicated...

```
x <- 5
MyFunc2 <- function(y){
   print(y)
   x <- y
   print(x)
}
> MyFunc2(10)
[1] 10
[1] 10
> x
[1] 5
```

- As a rule, anything created in a function **stays** within the function. So, when we assign `x <- y` you can consider this to be totally independent to what's happening outside of the function (i.e. in the working directory).

Return

- The return step clarifies what exactly the output of the function is. If there is only one line in the function, then the return is implied. So, the first example could be rewritten as

```
skewness<-function(x){
          mean((x-mean(x))^3)/(sqrt(var(x)))^3
}
```

  with the same result.

- As the number of steps increase, however, it is a good idea to explicitly specify what is returned.

Rules of thumb

- Some rules of thumb for functions:
  - Any data/information needed in a function should be passed in as arguments - do not rely on R going outside of the function to find it;
  - Using distinct names within a function has the added benefit of removing any ambiguity;
  - Any data to be returned from the function should be within the `return` command.

  With functions, 95% of the time it will do what you're expecting; if you follow the above rules then the remaining 5% will be taken care of as well.

Logical Constructs

- The basic idea
- Using logical constructs on vectors and matrices
- The if statement

The basic idea

- Logical constructs are the type that ask questions like: is x greater than y? and always have an answer of either True or False.

- Consider the following example:

```
> x <- 5
> y <- 6
> x > y
[1] FALSE
> x < y
[1] TRUE
```

- What R is doing here is comparing the two (scalar) objects, and returning either a TRUE or a FALSE, depending on the veracity of the comparison.

The basic idea *cont.*

- Similarly, at a vector level

```
> x <- 1:5
> x > 3
[1] FALSE FALSE FALSE  TRUE  TRUE
```

- R compares each element of the vector, and returns a TRUE or FALSE for each element. i.e.

$$\begin{pmatrix}1\\2\\3\\4\\5\end{pmatrix} > 3 = \begin{pmatrix}1 > 3\\2 > 3\\3 > 3\\4 > 3\\5 > 3\end{pmatrix} = \begin{pmatrix}F\\F\\F\\T\\T\end{pmatrix}$$

The basic idea *cont.*

- Finally, at a matrix level

```
> x <- 1:5
> y <- 5:1
> x > y
[1] FALSE FALSE FALSE  TRUE  TRUE
```

R compares each element in the vector `x` with its corresponding element in vector `y`. i.e.

$$\begin{pmatrix}1\\2\\3\\4\\5\end{pmatrix} > \begin{pmatrix}5\\4\\3\\2\\1\end{pmatrix} = \begin{pmatrix}1>5\\2>4\\3>3\\4>2\\5>1\end{pmatrix} = \begin{pmatrix}F\\F\\F\\T\\T\end{pmatrix}$$

The basic idea *cont.*

- Here we used the greater than sign. Other valid comparisons include:

`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to
`==`	Equal to

Using logical constructs on vectors and matrices

- So, why is this useful? Well, consider this - remember where we selected rows from a matrix using its row number; well you can also select rows using trues and falses. For example

```
> x <- matrix(1:6, nrow=3)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> x[c(T,F,T),]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

- We can generate the trues and falses by a comparison as well:

```
> x[ x[,1] != 2, ]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

- We have a very effective (and as it turns out efficient) means of selecting a subset of our dataset.

- Furthermore, once the dataset has been subsetted, you can assign values. For example (continuing from before)

```
> x[ x[,1]==1, 1] <- 99
> x
     [,1] [,2]
[1,]   99    4
[2,]    2    5
[3,]    3    6
```

Exercise: With the dataset `monogamy`, plot the monogamous values (using pch=1), and the polomogous values using `points` (and pch=3)

```
plot(monogamy[monogamy[,3]=="M", 1],
       monogamy[monogamy[,3]=="M", 2],
       type="p", pch=1)
points(monogamy[monogamy[,3]=="P", 1],
         monogamy[monogamy[,3]=="P", 2],
         pch=3)
```

The if command

- The `if` command has the following syntax:

  `if (`*condition*`) {`
  steps if condition true
  `}`
  `else {`
  steps if condition false
  `}`

- Note that in this case, the *condition* is a logical construct that should result in a single True or False (and not multiple ones, like the comparison of two vectors). E.g.

```
x <- 5
y <- 6
if (x < y) {
   print("Hi")
}
```

- Obviously,

```
x <- 1:5
y <- 5:1
if (x > y) {
   print("Hi")
}
```

  makes no sense, since `x > y` produces a combination of trues and falses, and it is not clear which one applies to the condition (it just uses the first one).

- However,

```
x <- 1:5
y <- 5:1
if (sum(x) > sum(y)) {
   print("Hi")
}
```

  is fine.

Looping

- The for command
- Bootstrapping
- When to loop, and when not to loop

The `for` command

- A `for` loop performs an operation a set number of times, incrementing a "counter" each time it is performed.

- The syntax is
  `for (`*object* `in` *vector*`){`
  do things with object
  `}`

- For example

  ```
  for (counter in 1:20) {
      print(counter)
  }
  ```

  would print every number from 1 to 20 and is equivalent to

  ```
  counter <- 1
  print(counter)
  counter <- 2
  print(counter)
  ...
  ```

The `for` command *cont.*

- Note that the object, in this case `counter`, "physically" takes on the incremented value for each loop.

- However, like with functions, be careful what you name your object, as you will write over any object of the same name in the working directory. For example

```
> i <- 0
> for (i in 1:20){}
> i
[1] 20
```

- The vector doesn't have to be a contiguous sequence. For example:

```
for (i in c(1,3,5,7)){
     print(i)
}
```

  is fine.

Exercise Find all multiples of 3 between 1 and 100

Hint: loop through, and divide each number by 3. Then `print` that number if `i/3 - round(i/3)` is equal to zero.

```
for (i in 1:100){
    if (i/3 - round(i/3)== 0)
       print(i)
}
```

The Bootstrap

A common statistical process is to repeat a random sample *n* times, and retain the output from each iteration. An example of this is called bootstrapping.

A bootstrap is a tool for estimating a statistic where it is theoretically difficult to do so. In this case, we'll bootstrap on the median of Length in the blush dataset to find its standard deviation. There are three steps:

1. take a random sample, with replacement (to do this, we use the command `sample`);

2. calculate the median of the sample and retain it;

3. repeat 1,000 times;

4. calculate the standard deviation of the retained values.

The Bootstrap *cont.*

The code to do this is

```
> outputs <- rep(NA, 1000)
# create a vector to store the medians in
> for (i in 1:1000){
+    outputs[i] <- median(sample(blush[,"Length"],
replace=T))
+ }
> sqrt(var(outputs))
[1] 0.4316818
```

Exercise:

1. Perform a bootstrap on the median Number in the blush dataset to find its standard deviation.

2. Turn this bootstrap into a function.

3. Again, turn this bootstrap into a function, but this time allow an argument that specifies how many times to run the loop.

```
> outputs <- rep(NA, 1000)
> for (i in 1:1000){
+     outputs[i] <- median(sample(blush[,"Number"],
      replace=T))
+ }
> sqrt(var(outputs))
[1] 0.1608676

MyBootstrap <- function(x){
      outputs <- rep(NA, 1000)
      for (i in 1:1000){
          outputs[i] <- median(sample(x,
          replace=T))
      }
      return(sqrt(var(outputs)))
}

MyBootstrap <- function(x,n){
      outputs <- rep(NA, n)
      for (i in 1:n)  outputs[i] <-
      median(sample(x, replace=T))
      return(sqrt(var(outputs)))
}
```

When not to loop

- There are many circumstances, like above, where looping is the best way forward. However, there are situations where it is also the most inefficient way as well. Here we'll look at a couple of these.

Example 1
You have a three column matrix (`x`),* and if the sum of the first two columns is greater than 10, you want the third column to be 1, otherwise it should be zero.

Slow way:

```
for (i in 1:1e5){
  if (x[i,1]+x[i,2] > 10)
     x[i,3] <- 1
  else
     x[i,3] <- 0
}
```

Time taken: 10 seconds

Fast way:

```
x[x[,1]+x[,2] > 10,3] <- 1
x[x[,1]+x[,2] <= 10,3] <- 0
```

Time taken: 0.6 seconds

*Generated by `x <- cbind(rpois(1e5, 0.5), rnorm(1e5), NA)`.

Example 2

Consider a matrix (`x`) where you wish to set any observations less than zero equal to zero. And to make it interesting, let the matrix have 1000 rows and 1000 columns (i.e. one million observations).*

Slow way:

```
x <- matrix(
for (i in 1:1000) {
    for (j in 1:1000) {
        if (x[i,j] < 0)
           x[i,j] <- 0
    }
}
```

Time taken: 34.04 seconds

Fast way:

```
x[ x<0 ] <- 0
```

Time taken: 0.98 seconds.

*This was generated using `x <- matrix(rnorm(1000*1000), nrow=1000)`.

Further development

- Using the help
- Book - Venables and Ripley

Using the help

- To start up the help system, type `help.start()` at the command line.
- Can either type `help(`*command*`)` at the command line, or else use the search engine on the HTML page.
- Each help page is broken into a number of sections.
  - Description
  - Usage
  - Arguments
  - Details
  - Value
  - See Also
  - Examples

The book

Venables, W.N. & Ripley, B.D.
"Modern Applied Statistics with S",
4th edn, Springer 2002