Data Manipulation and Visualisation: An R primer

Mark Dunning and Mike Smith

Contents

Introduction

Document last updated : 2014-12-15 11:18:57

This document will guide you through the key concepts and techniques required to manipulate and process data in R. It is intended to be a reference document that you can refer to when you need to perform a particular task in R. We have also created an R package to accompany this manual and automate the creation of several graphs.

The crukCIMisc package

We have created an R package `crukCIMisc` to accompany this manual. The source code is available on github and can be installed in the following manner (you will need to install the `devtools` package as shown below)

```
install.packages("devtools")
library(devtools)
install_github(repo = "crukCIMisc", username = "markdunning")
library(crukCIMisc)
```

Getting help with R

You are of course welcome to email us, mark.dunning@cruk.cam.ac.uk and mike.smith@cruk.cam.ac.uk, with any questions about R and the contents of this manual. However, we encourage you to use the online support forum (similar to the popular stackoverflow) that the Bioinformatics Core has installed. Go to;

http://bioinf-qa001/

If you get stuck on something in R, there is a good chance that someone else has experienced the same problem before. Tracking such problems will allow us to build-up a useful resource of commonly-asked questions, and solutions.

Recommended references

Much freely-available documentation about R is available from CRAN. In particular, the R reference card is recommended.

There is no shortage of other R tutorials available online. Some that have provided particular inspiration in the construction of this document are listed below:

- R cookbook
- Quick-R
- UC Riverside R & Bioconductor manual

Running R and RStudio

Our recommended way of using R is via the RStudio interface. Like R, this software is freely to download and available on any operating system. You will need to download the latest versions of R and RStudio.

Installing R

Download links for R can be found on the CRAN page (Comprehensive R Archive Network). There are separate downloads for Windows and Mac OSX. Unix, or advanced users will need to find the appropriate link to their distribuion, or consider downloading the source code.

Installing RStudio

RStudio was created by a third-party company and available as a free download. You can find an *installer* for your platform on the download page. Make sure that you are downloading the **Desktop** version of RStudio, rather than the *server* version.

About this document

This document was written in RStudio using the **markdown** format. If you see a section like this, you can copy-and-paste the text into RStudio and execute it.

```
print("Hello World")
```

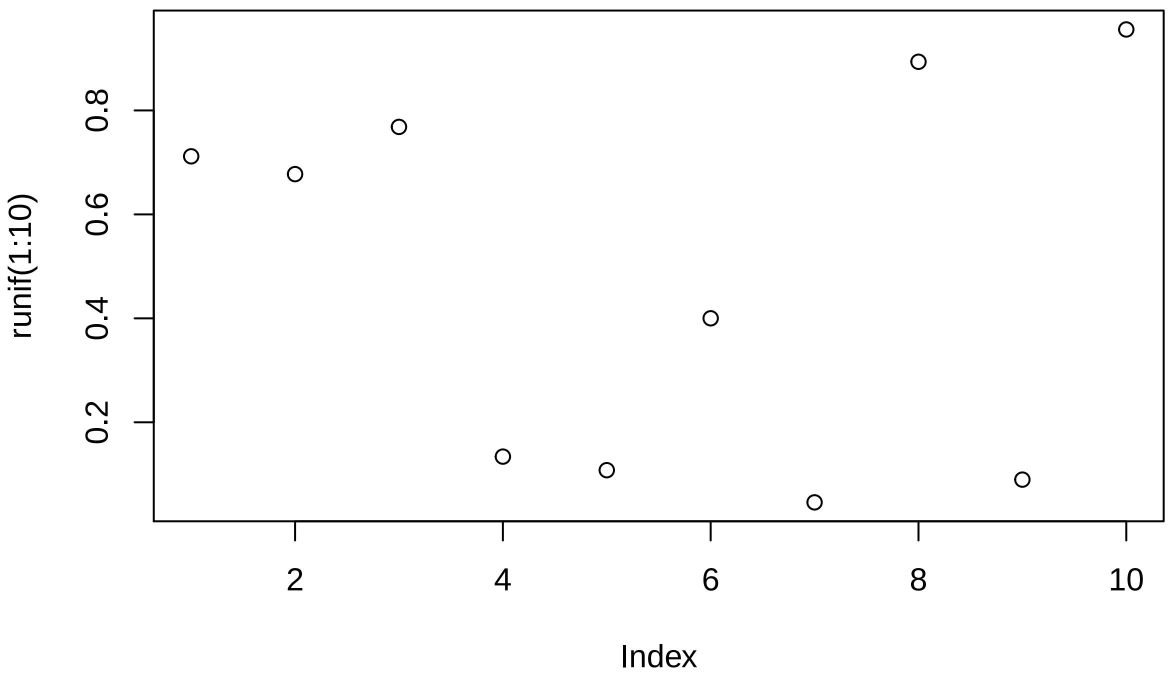
Often the output from R will be displayed after the R command. You can compare this to the output that you get from running the command yourself.

```r
print("Hello World")
```

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

Sometimes the graphics produced by R will be displayed too.

```r
plot(runif(1:10))
```



Reading data into R

The first stage of doing an analysis in R is to import some raw data that is presented a *spreadsheet-like* format. By which we mean data organised in rows and columns like we usually see in a spreadsheet program such a Microsoft Excel.

Unfortunately, importing data can be the source of many errors and frustrations; sometimes halting an analysis before it has even begun. *Well-formatted* data should be easy to read using a small selection of functions, although the user will have to decide which function to use. The key first steps are to locate the relevant raw data file on your hard drive and decide what format it is in. As a *rule-of-thumb*, tab-delimited files have file extensions *.txt* or *.tsv*, whereas comma-separated have an extension *.csv* .

More information on reading files into R can be found online at the R data manual or an introduction to data cleaning with R

Specify a path to the file

R needs to know where the file is located on your hard drive in order to read it. If the file is located within your *working directory* then you can just specify the name of the file. You can find out the location of your working directory using the `getwd` function.

```r
getwd()
```

You can also interrogate the contents of your working directory using the `dir` function

```r
dir()
```

The `dir` function has a useful argument `pattern` which will list files that contain a particular *string* of text. For instance, we can list all files of type `.csv` and `.txt` in the current working directory that could potentially be read into R.

```r
dir(pattern=".csv")
dir(pattern=".txt")
```

If the file you want to read is not in your working directory, then you will need to specify thepath to the file. A useful function in this situation is `file.choose`, which will open a dialog box allowing you to navigate to the file. The location can be stored as a variable.

```r
myfile <- file.choose()
```

When specifying the file location manually, it has to giving within *quote marks*, as in the following example

```r
myfile <- "PrimerExamples/Indepentent two-sample t-test_v2_long.txt"
```

We will now discuss various ways of loading tabular data into R. The examples should cover most file types that you might encounter (e.g. csv, txt, tsv, xlsx files). If you have an example file that doesn't seem to fit any of these example, please let us know. Please note that R is not able to import GraphPad Prism files, as this file format is proprietary. You will have to use a commercial version of Prism to export the data table into a text-file.

Once a file has been read into R, we recommend that you **check the dimensions and contents**. Sometimes R will read a file without error, but may not have been able to interpret the structure of the data correctly, which will cause problems for further analysis.

What separator should be used?

```r
count.fields("PrimerExamples/Indepentent two-sample t-test_v2_long.txt",sep="\t")
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [36] 2 2 2 2 2 2
```

```r
count.fields("PrimerExamples/Indepentent two-sample t-test_v2_long.txt",sep=",")
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [36] 1 1 1 1 1 1
```

Tab-delimited file

As a *rule-of-thumb*, tab-delimited files have file extensions *.txt* or *.tsv*. If your file is tab-delimited and you have specified the location using the previous section, you will be able to use the `read.delim` function. The `file.exists` is a useful function for checking that the file location is correct; returning `TRUE` if so.

```
myfile <- "PrimerExamples/Independent two-sample t-test_v2_long.txt"
file.exists(myfile)
```

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

```
mydata <- read.delim(myfile)
```

Assuming that the function does not produce an error, you should now have a variable `mydata` in your R workspace. This is *data frame* object, described in more detail in the appendix. At this point, it is highly recommended that you check the contents of the variable. The `View` command should open a new tab in RStudio.

```
View(mydata)
```

The dimensions and structure of the data can be further interrogated using the following useful functions.

```
dim(mydata)
```

```
## [1] 40  2
```

```
str(mydata)
```

```
## 'data.frame':    40 obs. of  2 variables:
##  $ Weight: num  20.77 9.08 9.8 8.13 16.54 ...
##  $ Breed : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
```

```
head(mydata)
```

```
##   Weight Breed
## 1  20.77     A
## 2   9.08     A
## 3   9.80     A
## 4   8.13     A
## 5  16.54     A
## 6  11.36     A
```

In this case, the object has 2 columns and 40 rows; as we would expect.

Now notice what happens if we read the file by (incorrectly) calling the `read.csv` function on a *tab-delimited* file

```
myfile <- "PrimerExamples/Independent two-sample t-test_v2_long.txt"
mydata <- read.csv(myfile)
dim(mydata)
```

```
## [1] 40  1
```

```r
str(mydata)
```

```
## 'data.frame':    40 obs. of  1 variable:
##  $ Weight.Breed: Factor w/ 39 levels "10.24\tA","10.28\tB",..: 27 34 39 31 23 5 6 9 17 24 ...
```

```r
head(mydata)
```

```
##   Weight.Breed
## 1     20.77\tA
## 2      9.08\tA
## 3       9.8\tA
## 4      8.13\tA
## 5     16.54\tA
## 6     11.36\tA
```

R was able to read the file without error, but the dimensions of the resulting object are incorrect. In particular, it has been unable to separate the columns because an incorrect s eparator was used (a `,` rather than a tab).

Comma-separated file

Files that have columns separated by commas can be read using the `read.csv` function. The file location needs to be specified by either using the `file.choose` function as follows;

```r
myfile <- file.choose()
```

Or by specifying the path to the file within quote marks. Once the file has been read, we check the dimensions and general structure.

```r
myfile <- "PrimerExamples/One-sample t-test.csv"
mydata <- read.csv(myfile)

dim(mydata)
```

```
## [1] 12  2
```

```r
str(mydata)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ Month       : Factor w/ 12 levels "April","August",..: 5 4 8 1 9 7 6 2 12 11 ...
##  $ Failure.rate: num  2.9 2.99 2.48 1.48 2.71 4.17 3.74 3.04 1.23 2.72 ...
```

```r
head(mydata)
```

```
##      Month Failure.rate
## 1  January         2.90
## 2 February         2.99
## 3    March         2.48
## 4    April         1.48
## 5      May         2.71
## 6     June         4.17
```

Excel file

R is even able to read Excel spreadsheets. This requires an additional *package* to be installed; `gdata`. This package is loaded as follows;

```r
library(gdata)
```

```
## gdata: read.xls support for 'XLS' (Excel 97-2004) files ENABLED.
##
## gdata: read.xls support for 'XLSX' (Excel 2007+) files ENABLED.
##
## Attaching package: 'gdata'
##
## The following object is masked from 'package:stats':
##
##     nobs
##
## The following object is masked from 'package:utils':
##
##     object.size
```

It is not uncommon for a package to print text to the screen upon loading. You should not mistake this text for an *error* message.

However, if you do not have `gdata` installed you will get an error message in the following form.

```
## Error in library(gdata) : there is no package called 'gdata'
```

If this error occurs, the package will need to be installed and then loaded

```r
install.packages("gdata")
library(gdata)
```

Note that you will only need to install the package once for a particular R *version*. However, you will need to load this library in every new R session when you require the functionality to read Excel files.

We can now call the `read.xls` function from `gdata` to create a *data frame* from your input file.

```r
myfile <- "PrimerExamples//Indepentent two-sample t-test_v2_long.xlsx"

mydata <- read.xls(myfile)
dim(mydata)
```

```
## [1] 40  2
```

```r
head(mydata)
```

```
##               Weight Breed
## 1 20.770000000000000     A
## 2  9.080000000000000     A
## 3  9.800000000000001     A
## 4  8.130000000000001     A
## 5 16.539999999999999     A
## 6 11.359999999999999     A
```

Specifying a different separator

If your file is neither `tab` or `comma` separated, then `read.csv` and `read.delim` will not be able to read the file correctly with their default arguments. In the following example, we try and use to `read.csv` on a file with columns that are separated with a space (`" "`).

```
myfile <- "PrimerExamples/space-separated.csv"
mydata <- read.csv(myfile)
dim(mydata)
```

```
## [1] 12  1
```

```
head(mydata)
```

```
##   Month.Failure.rate
## 1        January 2.9
## 2      February 2.99
## 3         March 2.48
## 4         April 1.48
## 5           May 2.71
## 6          June 4.17
```

We see that the dimensions of the data object are incorrect. Both `read.csv` and `read.delim` have the option to specify a different separator.

```
mydata <- read.csv(myfile, sep=" ")
dim(mydata)
```

```
## [1] 12  2
```

```
head(mydata)
```

```
##      Month Failure.rate
## 1  January         2.90
## 2 February         2.99
## 3    March         2.48
## 4    April         1.48
## 5      May         2.71
## 6     June         4.17
```

SPSS file

R is able to read data created by SPSS. This requires an additional *package* to be loaded; `foreign`.

```
library(foreign)
```

You only need to load the library *once* per R-session. We can now call the `read.spss` function from `foreign` to create a *data frame* from your input file.

```r
myfile <- "PrimerExamples/p004.sav"

mydata <- read.spss(myfile,to.data.frame = TRUE)
dim(mydata)
```

```
## [1] 199   7
```

```r
head(mydata)
```

```
##   CURRENTM PREVIOUS FAT PROTEIN DAYS LACTATIO I79
## 1       45       45 5.5     8.9   21        5   0
## 2       86       86 4.4     4.1   25        4   0
## 3       50       50 6.5     4.0   25        7   0
## 4       42       42 7.4     4.1   25        2   0
## 5       61       61 3.8     3.8   33        2   0
## 6       93       93 4.2     3.0   45        3   0
```

The `foreign` package is also able to read files in `SAS` (`read.export`) and `STATA` (`read.dta`) formats, but sadly not Graphpad Prism.

Further options for reading files

Sometimes we might be faced with 'messy' data that do not have a straightforward format of rows and columns. `read.csv` and `read.delim` are in fact specialised versions of a generic `read.table` function. This function has many options to allow finer-grain control over how files are read into R, as you will see from the manual page.

```r
?read.table
```

We now summarise the options that prove to be most useful

Skipping lines

By default, R will assume that your data table starts on the first line of the file and that the column headers are found on the first line. However, sometimes the data table can be preceeded with several lines of header information. The header may contain useful information for the user, such as the date the file was generated and software versions etc, but is not relevant when we try and read the data table into R.

If we try and use the default settings of `read.delim` to read a tab-delimited file with arbitrary header information, the following occurs. In this case, the file contains counts for a number of barcodes detected on a particular lane of sequencing. Naturally, we might want to read these data into R and perform some exploratory analysis and diagnostics.

```r
myfile <- "PrimerExamples/file-with-header.txt"
mydata <- read.delim(myfile)
head(mydata)
```

```
##   X100000_D00000_0000_C00CXXXXX
## 1  ============================
## 2             Lane 1 (SLX-0000)
## 3             -----------------
## 4                 260621866 reads
## 5             22313252 8.56% lost
## 6         1 = threshold for match
```

You will notice that R has tried to interpret the header information as lines of the data table, and therfore the data is not imported correctly.

Fortunately, both `read.csv` and `read.delim` have a `skip` argument so that we can start reading the data table from a pre-determined position in the file. In this example we have to *skip* the first 11 lines of the file in order to reach the table.

```
mydata <- read.delim(myfile, skip=11)
head(mydata)
```

```
##                Index    Total Balance        X    X0    X.1    X1
## 1 CGAGGCTG,CTCTCTAT 11772135  99.37% 11405699 4.37% 366436 0.14%
## 2 TAGGCATG,CTCTCTAT  9812302  82.82%  9515630 3.65% 296672 0.11%
## 3 CTCTCTAC,CTCTCTAT  8971207  75.72%  8715457 3.34% 255750 0.09%
## 4 CAGAGAGG,CTCTCTAT 13059606 110.24% 12650308 4.85% 409298 0.15%
## 5 GCTACGCT,CTCTCTAT 10757117  90.80% 10277334 3.94% 479783 0.18%
## 6 CGTACTAG,CTCTCTAT 14975852 126.41% 14462740 5.54% 513112 0.19%
```

In practice, you may have to try different values for the `skip` argument in order to obtain the correct result. Use of the `head` and `dim` functions should help you determine when the data has been read correctly.

Files with no header

Conversely, sometimes files can be created without any column names. By default, R assumes that the first row in the file contains column headings. This can have unexpected consequences when a file does not contain column headings.

```
myfile <- "PrimerExamples/file-with-no-header.csv"
mydata <- read.csv(myfile)
head(mydata)
```

```
##    January X2.9
## 1 February 2.99
## 2    March 2.48
## 3    April 1.48
## 4      May 2.71
## 5     June 4.17
## 6     July 3.74
```

```
dim(mydata)
```

```
## [1] 11  2
```

```
colnames(mydata)
```

```
## [1] "January" "X2.9"
```

In this example, R took the first line of the file to be the column headings and so we have column names January & X2.9. Moreover, the data frame produced has 11 rows rather than 12. The remedy is to set the `header` argument to `FALSE`. Both `read.csv` and `read.delim` are able to recognise the `header` argument.

```
myfile <- "PrimerExamples/file-with-no-header.csv"
mydata <- read.csv(myfile,header=FALSE)
head(mydata)
```

```
##         V1   V2
## 1  January 2.90
## 2 February 2.99
## 3    March 2.48
## 4    April 1.48
## 5      May 2.71
## 6     June 4.17
```

```
dim(mydata)
```

```
## [1] 12  2
```

```
colnames(mydata)
```

```
## [1] "V1" "V2"
```

You can modify the column names of the `mydata` object.

```
colnames(mydata) <- c("Month", "Value")
```

Reading a pre-determined number of lines

It is possible to read only a fixed number of rows by using the `nrows` argument. Useful if you just want to see the first few lines in a file.

```
subset <- read.csv("PrimerExamples/One-sample t-test.csv",nrows=4)
subset
```

```
##      Month Failure.rate
## 1  January         2.90
## 2 February         2.99
## 3    March         2.48
## 4    April         1.48
```

Invalid characters

Consider the following file, which we are told has the expression levels of 20 genes. We use `read.delim` to read the file (as it is tab-delimited)

```
myfile <- "PrimerExamples/DiffGenes.tsv"
mydata <- read.delim(myfile)
```

```
## Warning: EOF within quoted string
```

```
head(mydata)
```

```
##     LpR2 X3.57945023380892
## 1 fs(1)h             3.138
## 2 CG6954             2.749
## 3    Psa             2.701
## 4   zfh2             2.625
## 5   Fur1             2.441
## 6     ct             2.380
```

```
dim(mydata)
```

```
## [1] 11  2
```

However, the number of rows comes out as 11. We can use the `tail` function (analagous to `head`) to look at the last few rows in the file.

```
tail(mydata)
```

```
##
## 6
## 7
## 8
## 9
## 10
## 11 oc\t2.14209962314577\npros\t2.08816756905585\nKr-h1\t-2.04473631749177\nCG5149\t-2.15211774831633
##    X3.57945023380892
## 6              2.380
## 7              2.367
## 8              2.357
## 9              2.260
## 10             2.174
## 11                NA
```

Problems of this nature are usually due to an extraneous quotation mark " or ' being found somewhere in the file. We can tell `read.delim` to ignore quotation marks in the following way. Of course, we could have editted the raw data to remove the quote mark, but this might not be practical for larger files.

```
mydata <- read.delim(myfile,quote = "")
```

We now look at a second example file, which upon first impressions loads without any problems.

```
myfile <- "PrimerExamples/DiffGenes2.tsv"
mydata <- read.delim("PrimerExamples/DiffGenes2.tsv",header=FALSE)
head(mydata)
```

```
##        V1                V2
## 1     Psa 3.85289908747006
## 2     vnd 3.64569585507463
## 3      ct 3.20099018731361
## 4 fs(1)h 3.14893215463257
## 5     btd 3.12292776675935
## 6    zfh2 2.84206544957174
```

```
dim(mydata)
```

```
## [1] 27  2
```

A natural data exploration step would be to calculate the mean of the gene expression values, which are stored in the second column. However, we cannot do this even if we tell R to ignore any `NA` values. We can discover that the second has not been treated as numeric values by R.

```
mean(mydata[,2])
```

```
## Warning: argument is not numeric or logical: returning NA

## [1] NA
```

```
mean(mydata[,2],na.rm=TRUE)
```

```
## Warning: argument is not numeric or logical: returning NA

## [1] NA
```

```
summary(mydata[,2])
```

```
##                  ? -2.04041157013508  2.16058794530145  2.17521378431384
##                  1                 1                 1                 1
## -2.18074007124187  2.22322147868055  2.28021875859062   2.3040098655437
##                  1                 1                 1                 1
## -2.34805605180196  2.43003804114927 -2.44043700058667  2.51110827431386
##                  1                 1                 1                 1
##  2.54242322740185  2.54754537929952  2.56790148021494  2.60223270812433
##                  1                 1                 1                 1
## -2.72564011017977 -2.72929046084956  2.84206544957174  -2.9554891237748
##                  1                 1                 1                 1
##  3.12292776675935 -3.14132775338133  3.14893215463257  3.20099018731361
##                  1                 1                 1                 1
##  3.64569585507463  3.85289908747006  -3.8881615792709
##                  1                 1                 1
```

```
str(mydata)
```

```
## 'data.frame':    27 obs. of  2 variables:
##  $ V1: Factor w/ 27 levels "Awd","brat","btd",..: 20 26 10 11 3 27 21 19 4 23 ...
##  $ V2: Factor w/ 27 levels "?","-2.04041157013508",..: 26 25 24 23 21 19 16 15 14 13 ...
```

The issue is that our collaborator has chosen to use a `?` to denote a missing value in the second column. This causes R to think that this column contains characters rather than numbers. A solution is to use the `na.strings` argument to specify a particular value that is used to denote a missing value.

```
myfile <- "PrimerExamples/DiffGenes2.tsv"
mydata <- read.delim("PrimerExamples/DiffGenes2.tsv",header=FALSE,na.strings="?")

head(mydata)
```

```
##       V1    V2
## 1    Psa 3.853
## 2    vnd 3.646
## 3     ct 3.201
## 4 fs(1)h 3.149
## 5    btd 3.123
## 6   zfh2 2.842
```

```
str(mydata)
```

```
## 'data.frame':    27 obs. of  2 variables:
##  $ V1: Factor w/ 27 levels "Awd","brat","btd",..: 20 26 10 11 3 27 21 19 4 23 ...
##  $ V2: num  3.85 3.65 3.2 3.15 3.12 ...
```

```
summary(mydata[,2])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##  -3.890  -2.310   2.290   0.835   2.590   3.850       1
```

```
mean(mydata[,2],na.rm=TRUE)
```

```
## [1] 0.8349
```

If all else fails. . .

The `readLines` function can be used. This is a *low-level* function for reading data into R and reads each line in the file as avector, along with the characters used to separate the columns. For example, if we read a file with a `.txt` extension.

```
temp <- readLines("PrimerExamples/Indepentent two-sample t-test_v2_long.txt")
head(temp)
```

```
## [1] "Weight\tBreed" "20.77\tA"      "9.08\tA"       "9.8\tA"
## [5] "8.13\tA"       "16.54\tA"
```

The fact that a `\t` character can be seen in each item of the vector tells us that we should be using the `read.delim` function. Likewise, with a `.csv`

```r
temp <- readLines("PrimerExamples/One-sample t-test.csv")
head(temp)
```

```
## [1] "Month,Failure rate" "January,2.9"        "February,2.99"
## [4] "March,2.48"         "April,1.48"         "May,2.71"
```

each item in the vector has a `,` character which is used to separate the data into two columns. We should read this file using `read.csv`

If we have a file with `header` information, we can use `readLines` to identify the line on which the data table begins, and hence what we should use as a `skip` argument.

```r
temp <- readLines("PrimerExamples/file-with-header.txt")
temp[1:15]
```

```
##  [1] "100000_D00000_0000_C00CXXXXX"
##  [2] "============================"
##  [3] ""
##  [4] "Lane 1 (SLX-0000)"
##  [5] "------------------"
##  [6] ""
##  [7] "260621866 reads"
##  [8] "22313252 8.56% lost"
##  [9] "1 = threshold for match"
## [10] "4 = minimum distance between barcodes"
## [11] "Expected:"
## [12] "Index\tTotal\tBalance\t\t0\t\t1"
## [13] "CGAGGCTG,CTCTCTAT\t11772135\t99.37%\t11405699\t4.37%\t366436\t0.14%"
## [14] "TAGGCATG,CTCTCTAT\t9812302\t82.82%\t9515630\t3.65%\t296672\t0.11%"
## [15] "CTCTCTAC,CTCTCTAT\t8971207\t75.72%\t8715457\t3.34%\t255750\t0.09%"
```

In this case, we can see the column headings for the data table start on line 12. Thus, we should read the file with `skip=11` in the function `read.delim`.

Strings as factors

R is primarily a statistics language so much of its functionality is geared-up to make things ameanable to statistical analysis. For instance, it will prefer to work with Factors rather than characters. When a data table is read into R, as stored as a data frame, the default behaviour is to store character vectors as factors. This may not always be neccesary.

```r
myfile <- "PrimerExamples/One-sample t-test.csv"
mydata <- read.csv(myfile)
str(mydata)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ Month       : Factor w/ 12 levels "April","August",..: 5 4 8 1 9 7 6 2 12 11 ...
##  $ Failure.rate: num  2.9 2.99 2.48 1.48 2.71 4.17 3.74 3.04 1.23 2.72 ...
```

We do not plan to take advantage of this behaviour and can easily work with the months in character form. Therefore we can set the argument `stringsAsFactors=FALSE`

```r
myfile <- "PrimerExamples/One-sample t-test.csv"
mydata <- read.csv(myfile,stringsAsFactors=FALSE)
str(mydata)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ Month       : chr  "January" "February" "March" "April" ...
##  $ Failure.rate: num  2.9 2.99 2.48 1.48 2.71 4.17 3.74 3.04 1.23 2.72 ...
```

Data in multiple files

If the dataset you want to import is spread over several data files, you will have to read each file individually and combine the respective data frames into a single object.

Data from online sources

The `repmis` package (misellaneous tools for reproducible research) contains useful functions to retrieve online data.

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

A useful resource for such data is the Github page of Vincent Arel-Bundock, which has csv version of example datasets found in many R packages. We can download data from this page using the `source_data` function. For example, if we browse the web page we can see there is an dataset containing Ovarian Cancer survival data. We can copy the URL of the corresponding `.csv` file and paste into R.

```r
library(repmis)
surv <- source_data("http://vincentarelbundock.github.io/Rdatasets/csv/survival/ovarian.csv")
head(surv)
```

Using pre-built datasets

R comes with the `datasets` package which contains many example datasets ready for exploration and analysis. To see what datasets are available, along with brief descriptions, enter the following;

```r
data()
```

If you see a dataset that might be interesting (e.g. the `DNase` dataset which measures the density of a protein at various concentrations), it can be loaded into R in the following manner.

```r
data(DNase)
head(DNase)
```

```
##   Run    conc density
## 1   1 0.04883   0.017
## 2   1 0.04883   0.018
## 3   1 0.19531   0.121
## 4   1 0.19531   0.124
## 5   1 0.39062   0.206
## 6   1 0.39062   0.215
```

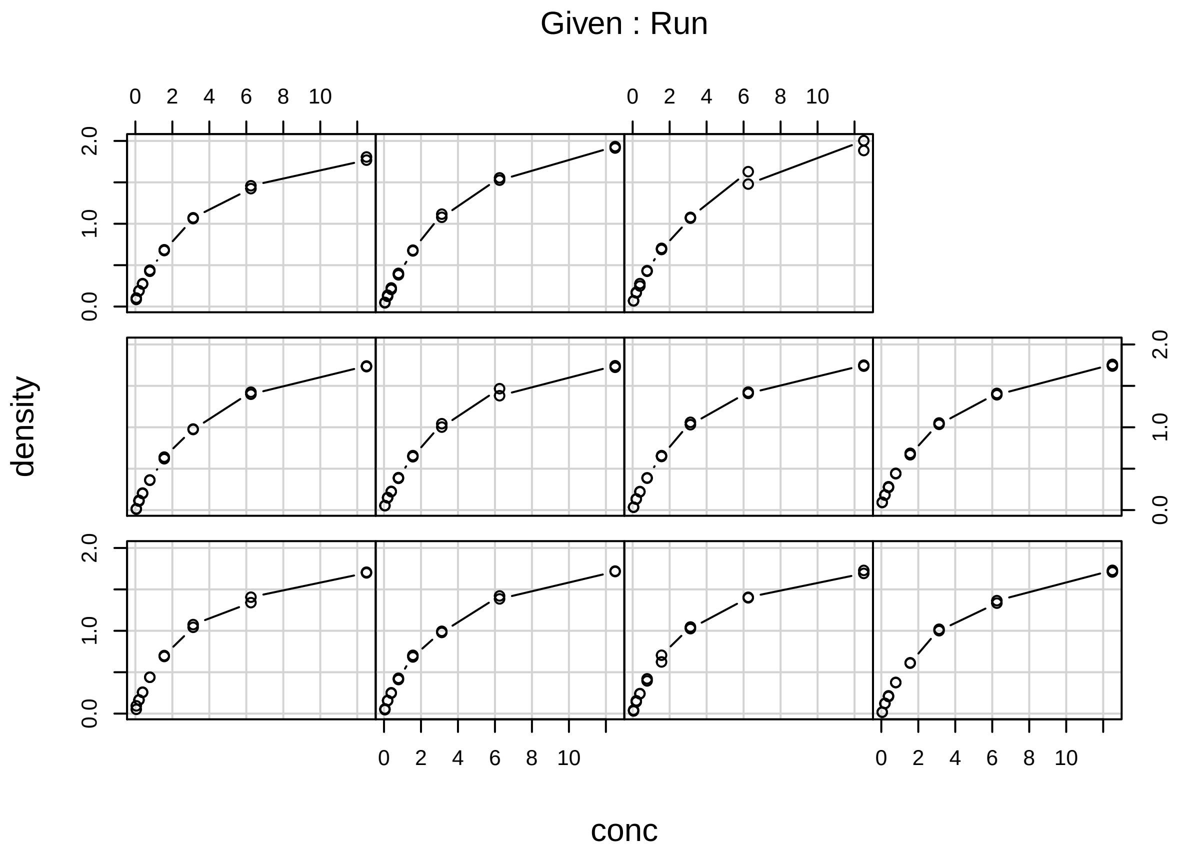
Further details on the dataset can be found on its help page.

```
?DNase
```

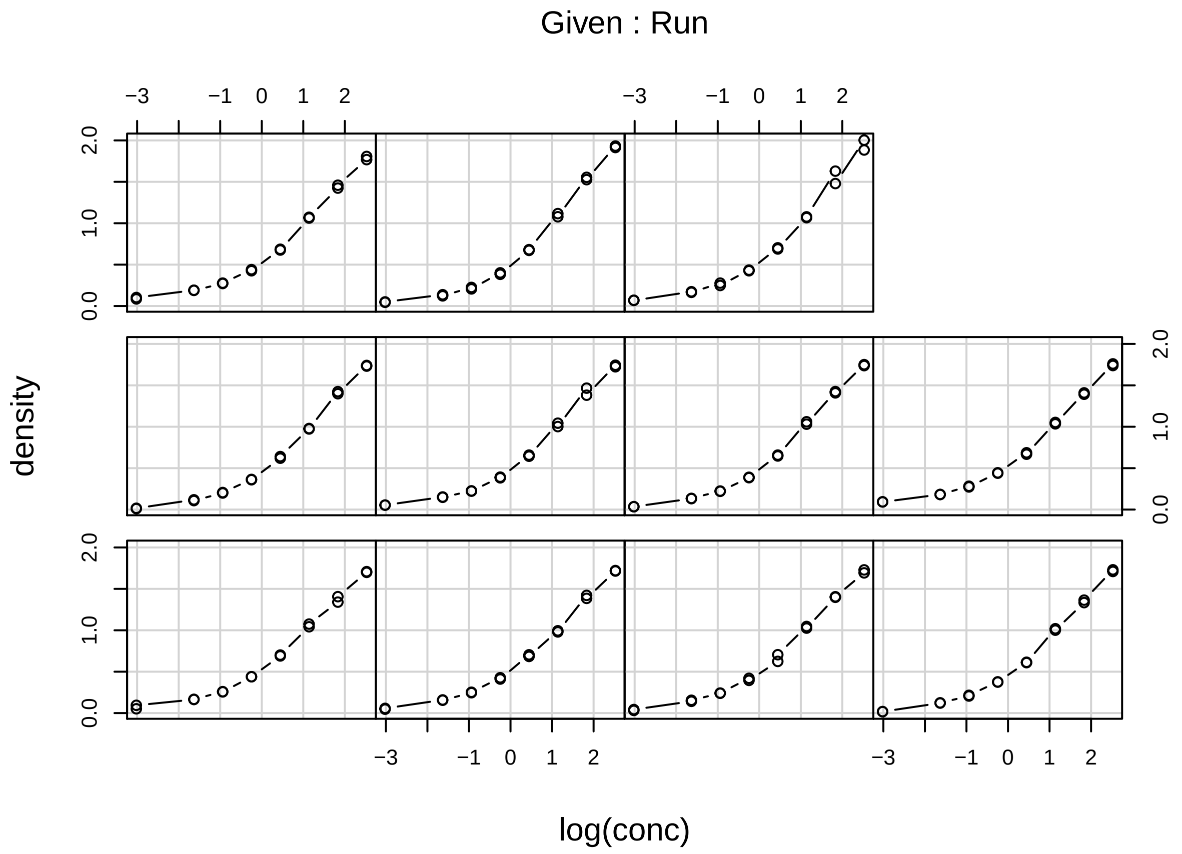
Moreover, example plots and code can be generated using the `example` function.

```
example(DNase)
```

```
##
## DNase> require(stats); require(graphics)
##
## DNase> ## Don't show:
## DNase> options(show.nls.convergence=FALSE)
##
## DNase> ## End Don't show
## DNase> coplot(density ~ conc | Run, data = DNase,
## DNase+        show.given = FALSE, type = "b")
```



```
##
## DNase> coplot(density ~ log(conc) | Run, data = DNase,
## DNase+        show.given = FALSE, type = "b")
```



```
##
## DNase> ## fit a representative run
## DNase> fm1 <- nls(density ~ SSlogis( log(conc), Asym, xmid, scal ),
## DNase+     data = DNase, subset = Run == 1)
##
## DNase> ## compare with a four-parameter logistic
## DNase> fm2 <- nls(density ~ SSfpl( log(conc), A, B, xmid, scal ),
## DNase+     data = DNase, subset = Run == 1)
##
## DNase> summary(fm2)
##
## Formula: density ~ SSfpl(log(conc), A, B, xmid, scal)
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## A     -0.0079     0.0172   -0.46     0.65
## B      2.3772     0.1095   21.71  5.4e-11 ***
## xmid   1.5074     0.1021   14.77  4.6e-09 ***
## scal   1.0626     0.0570   18.64  3.2e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0198 on 12 degrees of freedom
##
##
## DNase> anova(fm1, fm2)
## Analysis of Variance Table
```

```
##
## Model 1: density ~ SSlogis(log(conc), Asym, xmid, scal)
## Model 2: density ~ SSfpl(log(conc), A, B, xmid, scal)
##   Res.Df Res.Sum Sq Df   Sum Sq F value Pr(>F)
## 1     13    0.00479
## 2     12    0.00471  1 8.23e-05    0.21   0.66
```

Data manipulation

The term *data manipulation* is used here to describe the steps that occur after having read data into R and prior to plotting and/or statistical analysis.

Selecting columns from a data frame

We will illustrate using the `mtcars` data, one of the in-built datasets in R. We can load the data using the `data` function and query the dimensions and print the first few lines using `dim` and `head`.

Subsetting in R **always** uses the `[row,column]` notation.

By column number

```
data(mtcars)
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
dim(mtcars)
```

```
## [1] 32 11
```

Columns in the data frame can be retrived using a numeric index. The *first* column can be retrieved by specifying the *column* as 1 in `[row,column]` notation. However, we if we want values for all observations (rows) we can neglect to specify a value before the `,`. e.g.

```
mtcars[,1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

The output is a vector and it's length will be the same as the number of rows in the data frame. Note that we have not altered the `mtcars` data frame in any way. It will still have the same number of columns.

```
dim(mtcars)
```

```
## [1] 32 11
```

If we want to do some further calculations on the vector we have extracted, we need to assign it to a variable

```
myvector <- mtcars[,1]
length(myvector)
```

```
## [1] 32
```

```
myvector
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

The notation is very similar to retrieve the second column

```
myvector <- mtcars[,2]
length(myvector)
```

```
## [1] 32
```

```
myvector
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

If we want to retrieve the first and second columns *at the same time*, we can construct a vector, using the `c` function, comprising the indices we want to retrive and use this in the `[]` notation. The result is a *data frame* itself.

```
mydf <- mtcars[,c(1,2)]
head(mydf)
```

```
##                    mpg cyl
## Mazda RX4         21.0   6
## Mazda RX4 Wag     21.0   6
## Datsun 710        22.8   4
## Hornet 4 Drive    21.4   6
## Hornet Sportabout 18.7   8
## Valiant           18.1   6
```

```
dim(mydf)
```

```
## [1] 32  2
```

We can also use the `:` shortcut to generate a sequence of consecutive integers

```r
mydf <- mtcars[,c(1:3)]
head(mydf)
```

```
##                    mpg cyl disp
## Mazda RX4         21.0   6  160
## Mazda RX4 Wag     21.0   6  160
## Datsun 710        22.8   4  108
## Hornet 4 Drive    21.4   6  258
## Hornet Sportabout 18.7   8  360
## Valiant           18.1   6  225
```

```r
dim(mydf)
```

```
## [1] 32  3
```

By column name (Recommended)

We can also extract data using the names of columns. This is the recommended approach, as you might not be sure that the same variables will *always* be stored in the same column.

```r
myvec <- mtcars[,"mpg"]
head(myvec)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

The column names can be combined using the `c` function

```r
mydf <- mtcars[,c("mpg","cyl")]
head(mydf)
```

```
##                    mpg cyl
## Mazda RX4         21.0   6
## Mazda RX4 Wag     21.0   6
## Datsun 710        22.8   4
## Hornet 4 Drive    21.4   6
## Hornet Sportabout 18.7   8
## Valiant           18.1   6
```

We can also access columns by name using the `$` operator. By default, this will return all values in the column. This approach can be used in conjunction with *tab-completion*; meaning that you start typing the name of the column and press the `TAB` key. A list of possible column names will then appear.

```r
myvec <- mtcars$mpg
myvec
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Row subsetting

Subset rows by index

If we know the index of the row we want to retrieve, we can supply these indices in the [] notation. The get the first row, we can do

```
mtcars[1,]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

Muliple rows can be specified at a time. For example using the : shortcut

```
mtcars[1:5,]
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

Logical indexing

The term logical is used to describe values that are either TRUE or FALSE (sometimes called Boolean values in other languages). We can combine logical values into a vector and use for subsetting. In particular, they can be used to specify exactly which elements we extract from a vector. In this example, we create a vector of length 10, and another logical vector of the same length. Using the logical vector within the square brackets [] will tell R to extract elements *only* when the corresponding value in the the logical vector is TRUE

```
myvec <- 1:10
myvec
```

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

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

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

```
myvec[logvec]
```

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

In this case, logvec has TRUE values in positions 1,2,10. So these are the elements of myvec that can extracted in myvec[logvec].

There are several arithmetic operations that perform comparisons and produce a logical vector as output. The most common of these are >,<,==. Where == is used to test that two numerical values are the same.

If we wanted to restrict the data frame to rows where the `mpg` is less than `15`, we would first create a vector that will test all values of `mpg` to see if they are less than 15. Notice that the vector created is the same length as the number of rows in the data frame (32).

```
myind <- mtcars$mpg < 15
myind
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
length(myind)
```

```
## [1] 32
```

The logical vector, can now be used to subset the data frame. Only rows where `myind` has a value of `TRUE` will be returned.

```
mtcars[myind,]
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

Note that in the previous command we have created a *subset* of the data frame and not altered the original object; the number of rows is still 32

```
subset <- mtcars[myind,]
subset
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

```
dim(subset)
```

```
## [1]  5 11
```

```
dim(mtcars)
```

```
## [1] 32 11
```

Logical vectors can be used in combination, provided they are the same length. Now if we wanted all cars with `mpg < 15` *and* `drat > 3` we can combine the two conditions using the `&` operation.

```
myind <- mtcars$mpg < 15 & mtcars$drat > 3
myind
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
length(myind)
```

```
## [1] 32
```

Notice that `myind` is still length 32.

```
subset <- mtcars[myind,]
subset
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Duster 360        14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Camaro Z28        13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
```

Other ways of combining logical vectors include `|` (*or*) and `!` (*negation*).

%in%

A common task in exploratory data analysis is to ask whether a specific value or values exists in a larger R object. For example you may want to know whether a specific gene name appears in a long list of names you've just read in. To achieve this we can use the `%in%` command. Note that we have to specify each gene name within quotation marks `""`.

```
"BRCA1" %in% c("RAS", "MYC", "BRCA1", "TP53")
```

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

In the case above we were only interested in a single text query. The example below demonstrates both that `%in%` is equally happy working with numeric values, and that a vector of queries can be used in a single command. Here a vector of `TRUE` or `FALSE` values the same length as the number of queries will be returned. This is far more efficient that trying each in turn.

```
c(1,5,11) %in% 1:10
```

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

In our particular example, we could restrict the analysis to just rows that correspond to particular cars. The car names appear in the `rownames` of the data frame.

```
rownames(mtcars)
```

```
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
```

```
myInd <- rownames(mtcars) %in% c("Merc 230", "Merc 280")
myInd
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
mtcars[myInd,]
```

```
##           mpg cyl  disp  hp drat   wt qsec vs am gear carb
## Merc 230 22.8   4 140.8  95 3.92 3.15 22.9  1  0    4    2
## Merc 280 19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
```

`myInd` is a *logical* vector (i.e. either `TRUE` or `FALSE`) of length 32, where each denotes whether that particular rownames is one of the supplied vector. This logical vector can then be used to subset the data frame.

match,

While the `%in%` command is useful for telling you whether a query appears somewhere, it doesn't specify where in the subject you will find it. However there are several ways to achieve just this, where perhaps the most straightforward is `which()`.

```
which(c("RAS", "MYC", "BRCA1", "TP53") == "BRCA1")
```

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

However this approach falls down if one is interested in multiple queries.

```
which(c("RAS", "MYC", "BRCA1", "TP53") == c("BRCA1", "RAS"))
```

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

To avoid the confusion this could cause, it is better the employ the function `match()`

```
match(x = c("BRCA1", "MYC"), table = c("RAS", "MYC", "BRCA1", "TP53"))
```

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

If we just wanted to find which row corresponds to `Merc 230` we can use the `match` function to see the index of the `rownames(mtcars)` vector that corresponds to the *string* `"Merc 230"`.

```
myInd <- match("Merc 230", rownames(mtcars))
myInd
```

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

```
mtcars[myInd,]
```

```
##           mpg cyl  disp hp drat   wt qsec vs am gear carb
## Merc 230 22.8   4 140.8 95 3.92 3.15 22.9  1  0    4    2
```

However, we have to remember that `match` will only return the index of the *first* occurence. If the string we want to find occurs multiple times, then we may get unexpected results further on in the analysis. Also, the text has to *exactly* match. Sometimes it is safer to use the `grep` function.

grep

`grep` will return *all* rownames that contain the string `Merc` *somewhere* in their name we can use the `grep` function.

```
myInd <- grep("Merc", rownames(mtcars))
myInd
```

```
## [1]  8  9 10 11 12 13 14
```

```
mtcars[myInd,]
```

```
##              mpg cyl  disp  hp drat   wt qsec vs am gear carb
## Merc 240D   24.4   4 146.7  62 3.69 3.19 20.0  1  0    4    2
## Merc 230    22.8   4 140.8  95 3.92 3.15 22.9  1  0    4    2
## Merc 280    19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
## Merc 280C   17.8   6 167.6 123 3.92 3.44 18.9  1  0    4    4
## Merc 450SE  16.4   8 275.8 180 3.07 4.07 17.4  0  0    3    3
## Merc 450SL  17.3   8 275.8 180 3.07 3.73 17.6  0  0    3    3
## Merc 450SLC 15.2   8 275.8 180 3.07 3.78 18.0  0  0    3    3
```

We can perform much more complex queries (e.g. only searching for words *starting* with 'Merc', or finding specific patterns of numbers and letter) using a tool called "Regular Expression" or "Regex". However, this beyond the scope of this document but a comprehensive introduction can be found at http://www.zytrax.com/tech/web/regex.htm

Adding new columns / variables

Re-ordering data frames

Often we want to sort data based on one or more properties. R has several options for performing this type of operation.

First we'll look at the `order()` function. In its simplest form this function takes a list of values and returns their ordering in *ascending* value. So in our example below the 15th entry is the lowest, followed by the 16th, while the 20th entry is the highest.

If we actually want to put the data in this order, we need to use the subsetting approach.

```
order(mtcars[,"mpg"])
```

```
##  [1] 15 16 24  7 17 31 14 23 22 29 12 13 11  6  5 10 25 30  1  2  4 32 21
## [24]  3  9  8 27 26 19 28 18 20
```

```
mtcars[order(mtcars[,"mpg"]),]
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
```

To list the entries in the opposite order we can specify the argument `decreasing = TRUE`.

```
mtcars[order(mtcars[,"mpg"],decreasing=TRUE),]
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
```

Alternatively, R also includes the commnad `sort()`, which will perform the reordering automatically. It is also useful to note that both `order()` and `sort()` will work on character data too where items will be sorted alphabetically.

```
sort(mtcars[,"mpg"])
```

```
##  [1] 10.4 10.4 13.3 14.3 14.7 15.0 15.2 15.2 15.5 15.8 16.4 17.3 17.8 18.1
## [15] 18.7 19.2 19.2 19.7 21.0 21.0 21.4 21.4 21.5 22.8 22.8 24.4 26.0 27.3
## [29] 30.4 30.4 32.4 33.9
```

```
sort(rownames(mtcars))
```

```
##  [1] "AMC Javelin"         "Cadillac Fleetwood"  "Camaro Z28"
##  [4] "Chrysler Imperial"   "Datsun 710"          "Dodge Challenger"
##  [7] "Duster 360"          "Ferrari Dino"        "Fiat 128"
## [10] "Fiat X1-9"           "Ford Pantera L"      "Honda Civic"
## [13] "Hornet 4 Drive"      "Hornet Sportabout"   "Lincoln Continental"
## [16] "Lotus Europa"        "Maserati Bora"       "Mazda RX4"
## [19] "Mazda RX4 Wag"       "Merc 230"            "Merc 240D"
## [22] "Merc 280"            "Merc 280C"           "Merc 450SE"
## [25] "Merc 450SL"          "Merc 450SLC"         "Pontiac Firebird"
## [28] "Porsche 914-2"       "Toyota Corolla"      "Toyota Corona"
## [31] "Valiant"             "Volvo 142E"
```

Transpose a matrix

```
mydata <- read.csv("PrimerExamples/Paired two-sample t-test.csv")
dim(mydata)
```

```
## [1] 20  2
```

```
mydata.t <- t(mydata)
mydata.t
```

```
##             [,1] [,2]  [,3]  [,4]  [,5] [,6]  [,7] [,8] [,9] [,10] [,11]
## Ovarian    1201 1030 895.6 842.1 903.1 1312 833.5 1008 1466 967.8 812.7
## Peritoneal 1156 1021 881.2 830.8 897.1 1263 823.1  951 1451 978.1 778.3
##             [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20]
## Ovarian    884.1  1359  1280 942.4 884.3 930.1  1147 881.5  1315
## Peritoneal 823.6  1336  1294 925.8 891.3 892.0  1133 847.8  1338
```

Text manipulation

Since we often begin an anaylsis be reading data from text files or spreadsheets, one frequently finds they require tools to manipulate this text data in ways that make further analysis easier e.g. extracting componant parts from complex sample IDs or manipulating date and time information.

Finding the number of characters

```
nchar(rownames(mtcars))
```

```
##  [1]  9 13 10 14 17  7 10  9  8  8  9 10 10 11 18 19 17  8 11 14 13 16 11
## [24] 10 16  9 13 12 14 12 13 10
```

Changing case

Sometimes it can be useful to standardise the case of any character information you're working with. It is not uncommon to import data from different sources where alternative choices have been made regarding capitalisation. For example it is not hard to imagine reading two tables relating to the same experiment where in one the entries are labelled "SAMPLE N" and in the other "Sample N". It is much easier to try and match entries in these two tables if they are converted to the same format.

R includes the functions `toupper()` and `tolower()` that will perform this transformation for you, rather than needing to edit the data directly.

```
exampleNames <- c("Sample 1", "Sample 2", "Sample 3", "Sample 4")
toupper(exampleNames)
```

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

```
tolower(exampleNames)
```

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

Splitting into parts

The function `strsplit()` allows you to break a string into smaller parts, based on a specified criteria. In the example below we break our sample names where there is a space.

```
exampleNames <- c("Sample 1", "Sample 2", "Sample 3", "Sample 4")
strsplit(x = exampleNames, split = " ")
```

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

The output from this is a list, where the ith entry is the result of breaking the ith string passed to the function.

In cases like our example above, where we know every entry will be split into a consistent number of parts, it's often easier to transform this output into a matrix where each row represents an entry. To do this we make use of the `unlist()` function, which transforms our list into a vector followed by the command to create a matrix.

```
exampleNames <- c("Sample 1", "Sample 2", "Sample 3", "Sample 4")
namesList <- strsplit(x = exampleNames, split = " ")
matrix( unlist( namesList ), ncol = 2, byrow = TRUE )
```

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

Extracting substrings

If we only want to retain a portion of a string we can use the `substring` function. In addition to the strings we want to process this takes two arguments: `start` and `stop` which are integers specifying the first and last characters we want to retain. In the example below we retain the last 5 characters (note the space is treated just like any other character).

```
exampleNames <- c("Sample 1", "Sample 2", "Sample 3", "Sample 4")
substr(exampleNames, start = 4, stop = 8)
```

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

Replacing parts of strings

In addition to extracting portions of strings, you can replace them using the function `sub()`, much like one might use find and replace in a text editor. In the following example we start with 3 sample names. We then use the `sub()` command to replace "Person" with "Sample" and include a space before the number.

```
names <- c("Person1", "Person2", "Person3")
sub(pattern = "Person", replacement = "Sample ", x = names)
```

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

The us a second substitution function called `gsub()`. This works in much the same way as `sub()`, but replaces every instance of the pattern it finds, where as `sub()` only replaces the first occurance.

```
repeatExample <- c("Here Here")
sub(pattern = "Here", replacement = "Hear", x = repeatExample)
```

```
## [1] "Hear Here"
```

```
gsub(pattern = "Here", replacement = "Hear", x = repeatExample)
```

```
## [1] "Hear Hear"
```

Combining strings

To combine two strings we can use `paste()`. The result is a single string.

```
paste("Hello", "World")
```

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

We can combine more than two strings, as seen below. This example also highlights how shorter sets of strings (in this case "Month" and "is") will be reused as many times as necessary.

```
paste("Month", 1:12, "is", month.name)
```

```
##  [1] "Month 1 is January"   "Month 2 is February"  "Month 3 is March"
##  [4] "Month 4 is April"     "Month 5 is May"       "Month 6 is June"
##  [7] "Month 7 is July"      "Month 8 is August"    "Month 9 is September"
## [10] "Month 10 is October"  "Month 11 is November" "Month 12 is December"
```

The default behaviour of `paste()` is to place spaces between the string. If we want to alter this we need to specify the `sep` argument. Here we combine the months of the year with the number of characters in each word.

```
##  [1] "January:7"   "February:8"  "March:5"     "April:5"     "May:3"
##  [6] "June:4"      "July:4"      "August:6"    "September:9" "October:7"
## [11] "November:8"  "December:8"
```

Long versus wide format

Calculating summary statistics

```r
mydata <- matrix(runif(1000),ncol=5)
summary(mydata)
```

```
##        V1               V2               V3               V4
##  Min.   :0.0135   Min.   :0.0202   Min.   :0.0016   Min.   :0.0149
##  1st Qu.:0.2807   1st Qu.:0.2802   1st Qu.:0.2703   1st Qu.:0.2622
##  Median :0.4754   Median :0.5298   Median :0.4552   Median :0.4862
##  Mean   :0.5073   Mean   :0.5304   Mean   :0.4798   Mean   :0.4996
##  3rd Qu.:0.7580   3rd Qu.:0.7821   3rd Qu.:0.7224   3rd Qu.:0.7398
##  Max.   :0.9928   Max.   :0.9976   Max.   :0.9996   Max.   :0.9919
##        V5
##  Min.   :0.0007
##  1st Qu.:0.2606
##  Median :0.5326
##  Mean   :0.5162
##  3rd Qu.:0.7433
##  Max.   :0.9998
```

```r
colMeans(mydata)
```

```
## [1] 0.5073 0.5304 0.4798 0.4996 0.5162
```

```r
rowMeans(mydata)
```

```
##   [1] 0.6352 0.4974 0.5621 0.5554 0.3327 0.4685 0.3456 0.3695 0.4628 0.3971
##  [11] 0.6993 0.6260 0.3455 0.3589 0.6534 0.6222 0.3338 0.6381 0.3807 0.4602
##  [21] 0.5630 0.3686 0.5532 0.4896 0.6699 0.5674 0.4722 0.5031 0.5814 0.4096
##  [31] 0.5743 0.3545 0.3240 0.4430 0.5858 0.6559 0.4780 0.4064 0.7894 0.7948
##  [41] 0.4864 0.5183 0.6420 0.6670 0.6290 0.4114 0.5155 0.5841 0.4546 0.4832
##  [51] 0.6056 0.4692 0.5891 0.6203 0.1928 0.4540 0.5445 0.6039 0.6367 0.4912
##  [61] 0.4532 0.5350 0.3714 0.6608 0.6547 0.8046 0.4179 0.5655 0.5935 0.3735
##  [71] 0.6246 0.6341 0.5579 0.5066 0.4228 0.4545 0.5371 0.6332 0.4773 0.6210
##  [81] 0.5824 0.5725 0.4554 0.6412 0.3704 0.5139 0.6116 0.5275 0.6463 0.2939
##  [91] 0.4339 0.4720 0.5372 0.6112 0.5097 0.4469 0.3281 0.3906 0.7108 0.8021
## [101] 0.5717 0.4927 0.5273 0.4382 0.4618 0.5804 0.7585 0.6795 0.5051 0.4435
## [111] 0.4344 0.3999 0.5220 0.3655 0.1925 0.6342 0.2628 0.4834 0.6883 0.6777
## [121] 0.5914 0.5933 0.6659 0.5671 0.4801 0.4437 0.6130 0.5333 0.3799 0.5399
## [131] 0.3755 0.4177 0.5041 0.5387 0.3586 0.2401 0.2727 0.5021 0.3284 0.6635
## [141] 0.6084 0.2844 0.5298 0.4795 0.2847 0.5295 0.5381 0.6182 0.6613 0.4858
## [151] 0.3577 0.5588 0.3789 0.4624 0.7408 0.4281 0.5882 0.6188 0.4946 0.5235
## [161] 0.3778 0.5028 0.5077 0.4403 0.5150 0.4273 0.3924 0.4024 0.5427 0.4827
## [171] 0.5799 0.5764 0.4956 0.5236 0.2856 0.6098 0.4512 0.5912 0.4053 0.3738
## [181] 0.7745 0.4743 0.3243 0.5893 0.6405 0.6636 0.3734 0.4005 0.3460 0.4223
## [191] 0.6720 0.2741 0.4308 0.5304 0.2849 0.3393 0.5108 0.6245 0.4635 0.5268
```

```
IQR(mydata[,1])
```

```
## [1] 0.4773
```

```
sd(mydata[,1])
```

```
## [1] 0.2812
```

apply, tapply, aggregate

Writing data to a file

Plotting

A plot can be created in R simply by calling the `plot` function with a valid R object.

There are many optional arguments can be altered to change the apperance of the plots e.g adding titles or labels, altering the colour and shapes used to plot, or modifying the range of axes. Many of these options are available across a multitude of plotting functions. We'll initially demonstrate them using the histogram function but most can be applied to whatever type of plot you are trying to generate.

A histogram

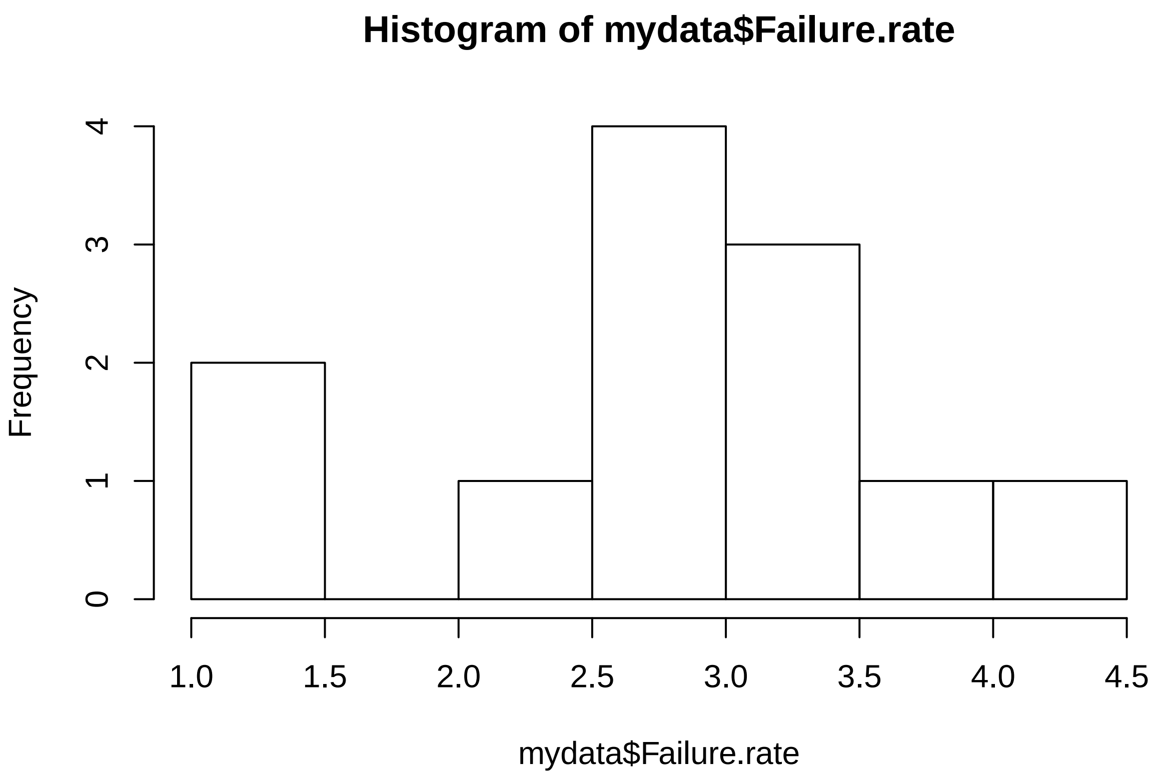
The `hist` function requires a *numeric* vector, which could be the result of some calculation you or have performed or the column from a data frame. In this example, we will read an example file

```
mydata <- read.csv("PrimerExamples/One-sample t-test.csv")
mydata
```

```
##        Month Failure.rate
## 1    January         2.90
## 2   February         2.99
## 3      March         2.48
## 4      April         1.48
## 5        May         2.71
## 6       June         4.17
## 7       July         3.74
## 8     August         3.04
## 9  September         1.23
## 10   October         2.72
## 11  November         3.23
## 12  December         3.40
```

The data concern (fictional) failure rates for microarrays in Genomics. To visualise the distribution of the failure rates, we could use a histogram using the `hist` function. We have to select the relevant column from the data frame, in this case the second column.

```r
hist(mydata$Failure.rate)
```

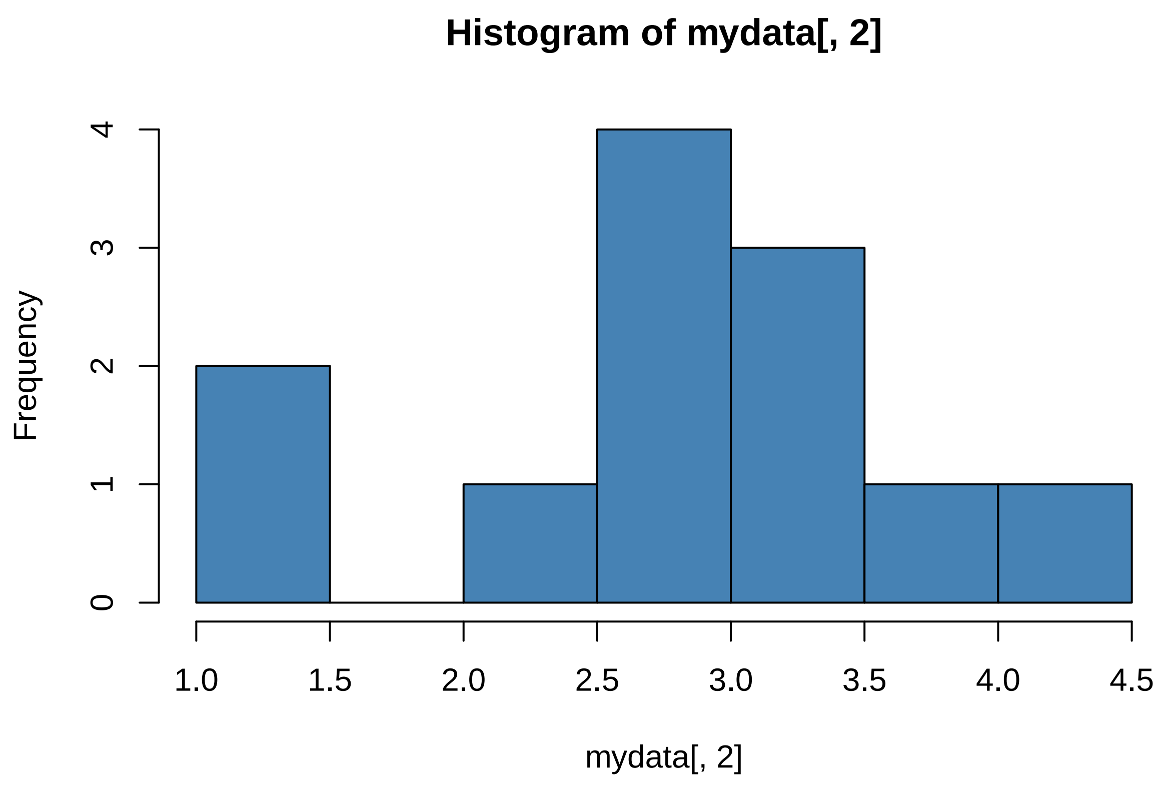


Unfortunately, the default plot that R creates is not that pretty. As with all plotting functions, there is a large amount of customisations that are possible. Details are on the *help* page for `hist`.

```r
?hist
```

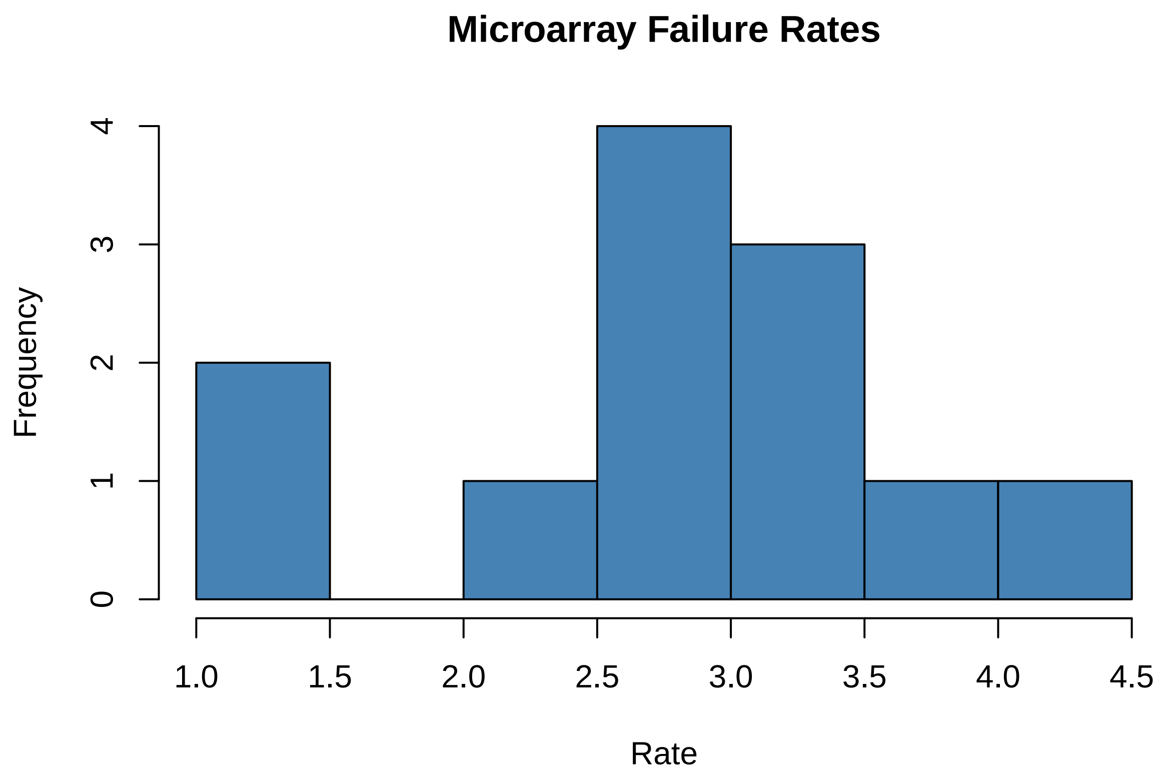
We can specify an alternative colour using the `col` argument.

```r
hist(mydata[,2],col="steelblue")
```



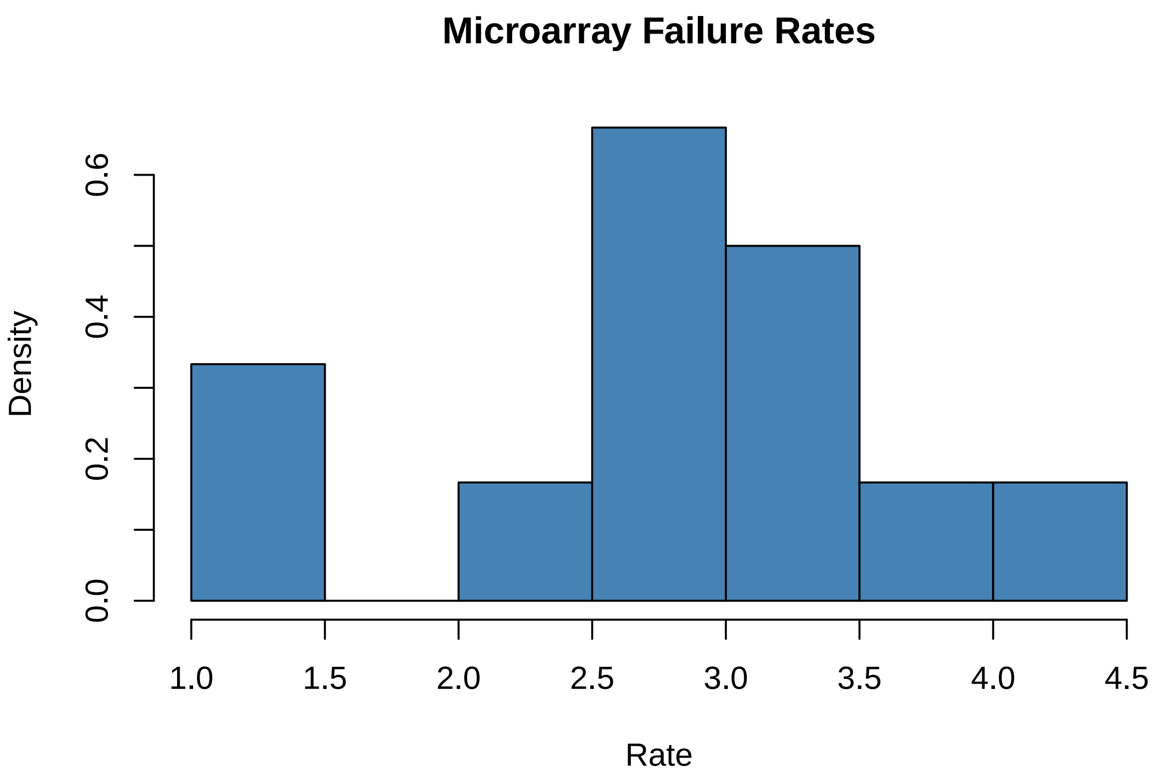
And add more informative labels and titles using `main` and `xlab`

```
hist(mydata[,2],col="steelblue",main="Microarray Failure Rates",xlab="Rate")
```



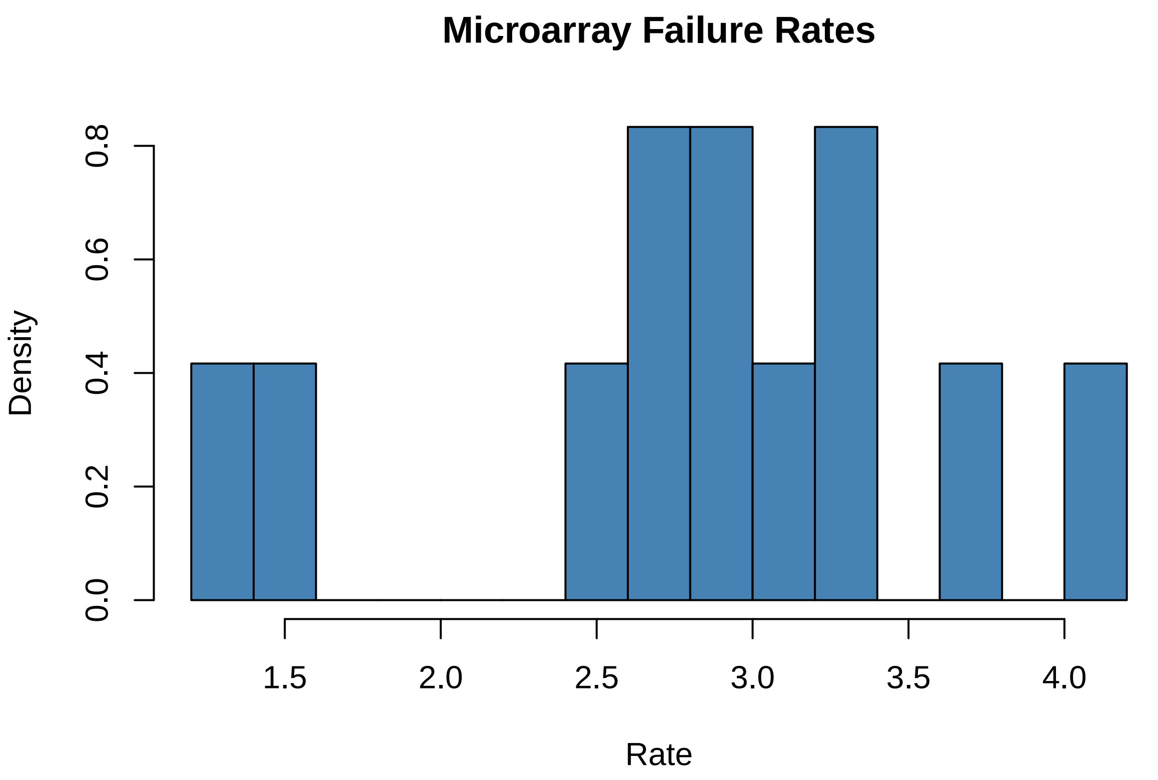
Technical modifications include displaying frequencies on the y-axis rather than counts.

```r
hist(mydata[,2],col="steelblue",main="Microarray Failure Rates",xlab="Rate",freq =  FALSE)
```



We can also specify the number of *breaks*

```r
hist(mydata[,2],col="steelblue",main="Microarray Failure Rates",xlab="Rate",freq =  FALSE,breaks = 12)
```



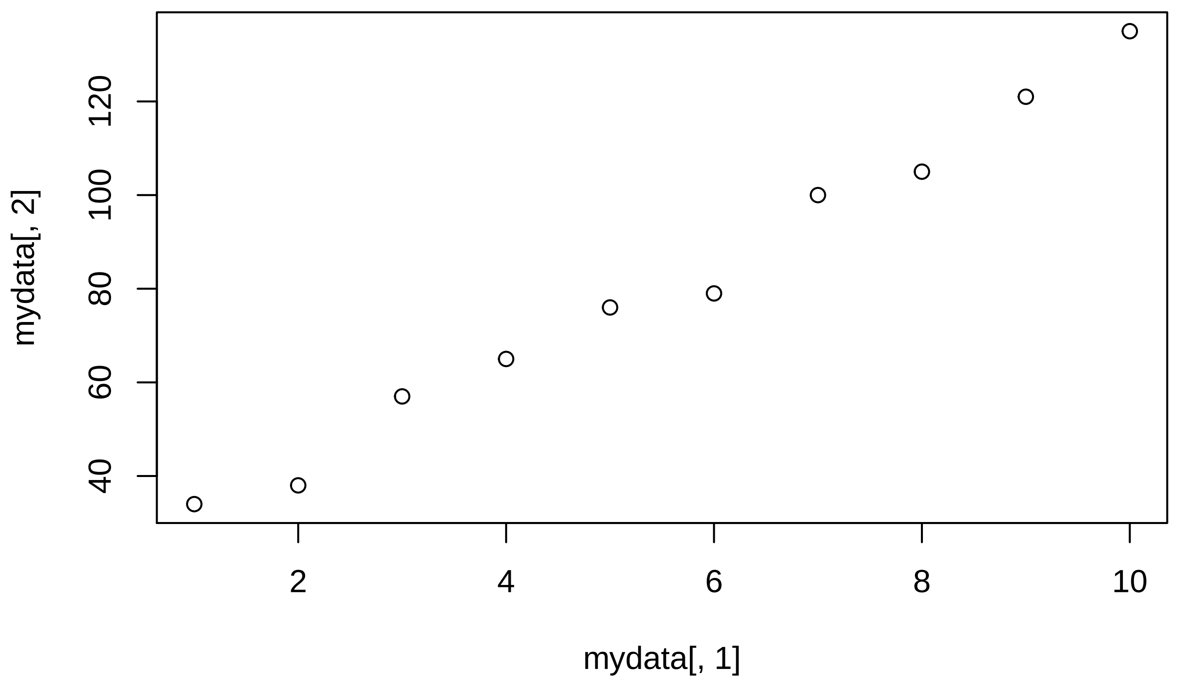
A scatter plot

The `plot` function is the generic function for plotting in R, and can produce different plots depending on the input. A scatter plot can be generated by specifying a *vector* to go along the x-axis, and a *vector* for the y axis. It is assumed that the first argument supplied is for the x axis, and the second for the y -axis.

```
mydata <- read.delim("PrimerExamples/Linear regression.txt")
mydata
```

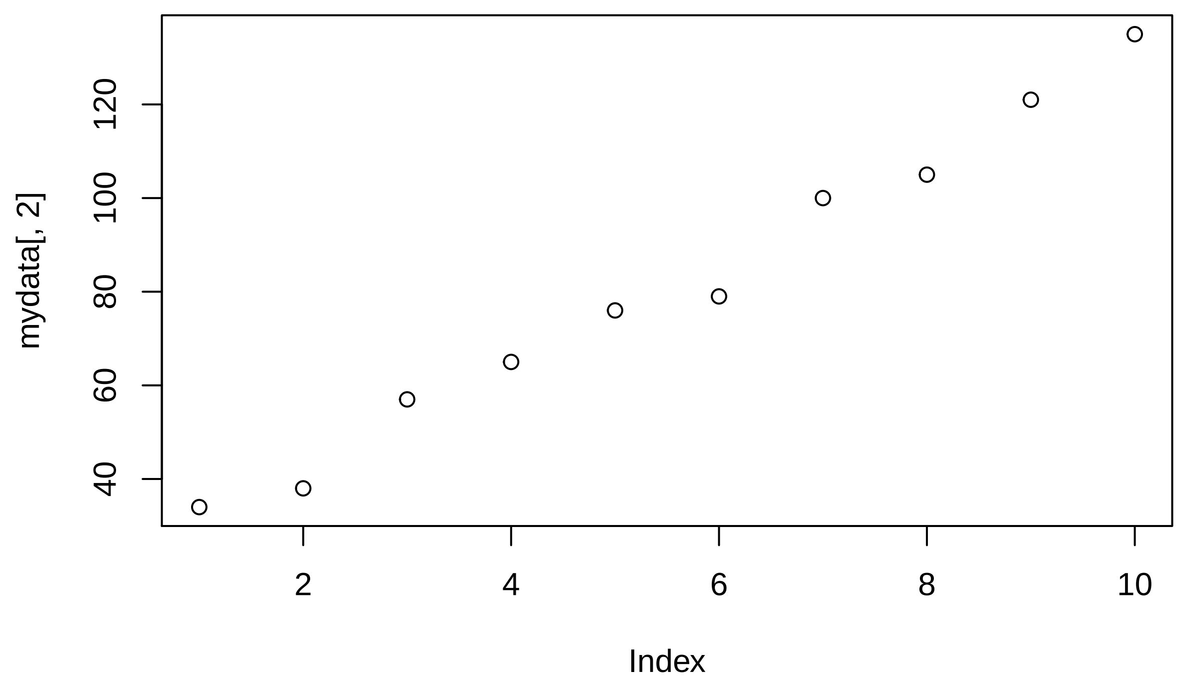
```
##    Minutes Control Control.1 Control.2 Treated Treated.1 Treated.2
## 1        1      34        29        28      31        29        44
## 2        2      38        49        53      61        NA        89
## 3        3      57        NA        55      78        99        77
## 4        4      65        65        50      93       111       109
## 5        5      76        91        84      NA       109       141
## 6        6      79        93        98     134       145       129
## 7        7     100       107        89     156       134       167
## 8        8     105       123       119     167        NA       180
## 9        9     121       143       134     178       192       175
## 10      10     135       156        NA     198       203       234
```

```
plot(mydata[,1],mydata[,2])
```



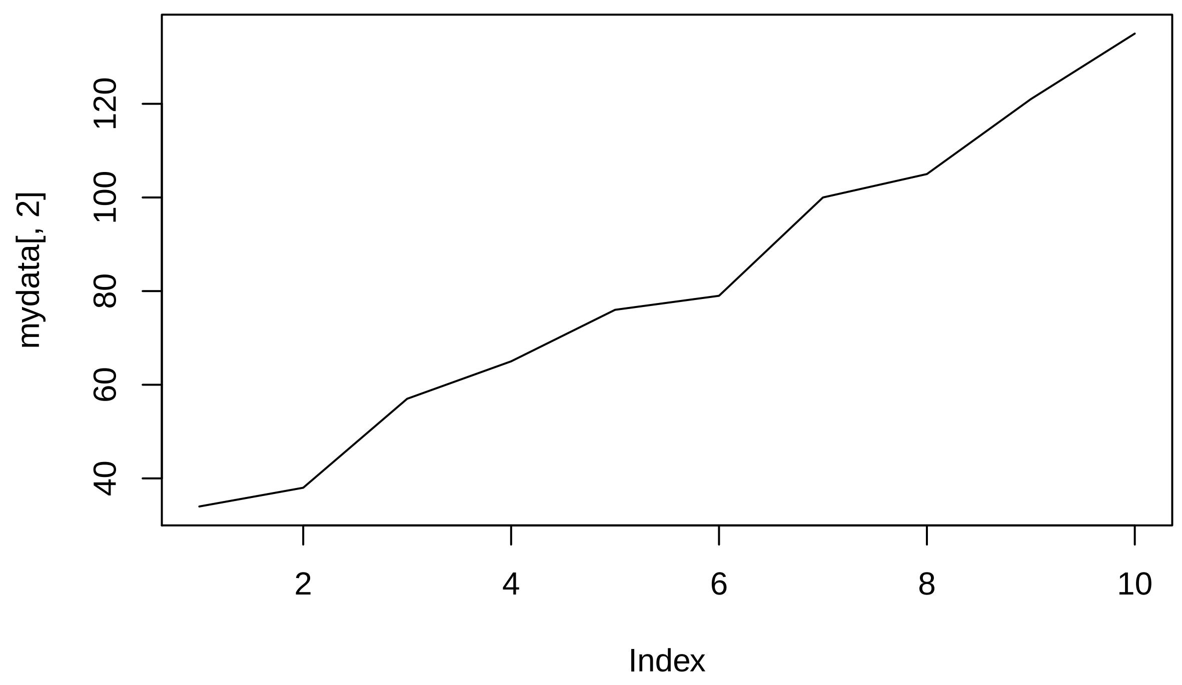
We can in fact just specify one vector, in which case the x axis is assumed to be the *index* of vector.

```
plot(mydata[,2])
```



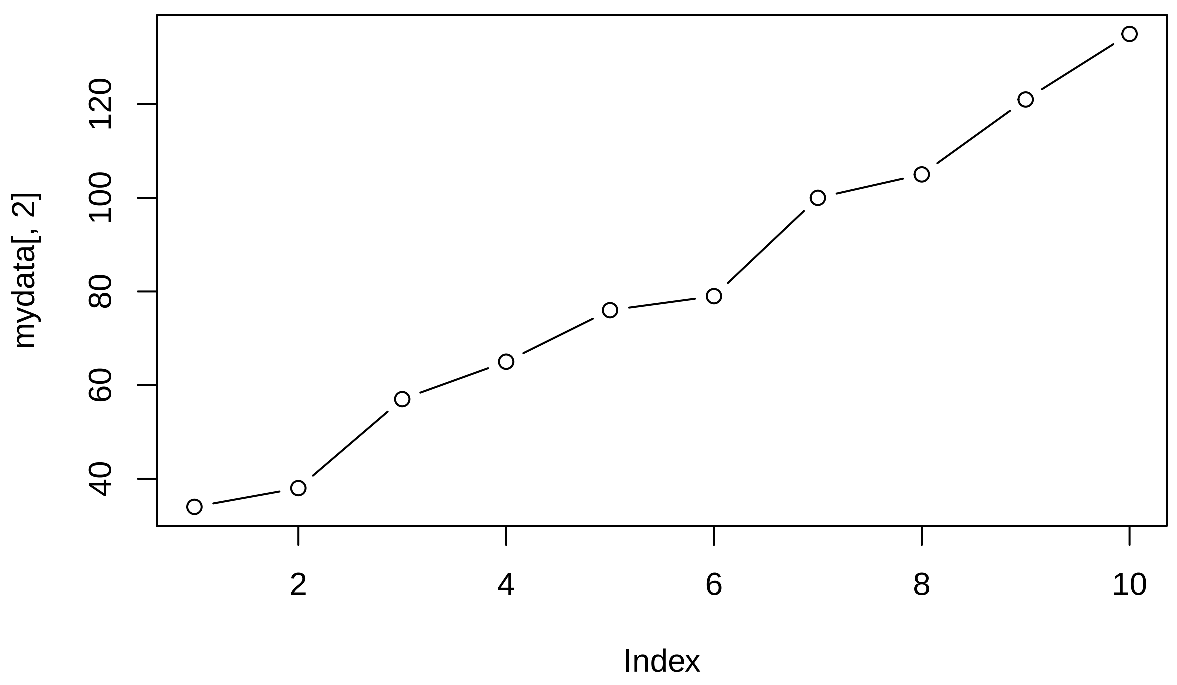
By default, a open circle is drawn at each x and y combination. If we wish, we can draw a *line* through the points by using the argument `type="l"`

```
plot(mydata[,2],type="l")
```



Or choose to have both a line and the points by specifying `type="b"`

```
plot(mydata[,2],type="b")
```



Later on, we will show how to add extra lines and points to this basic plot

Pie chart

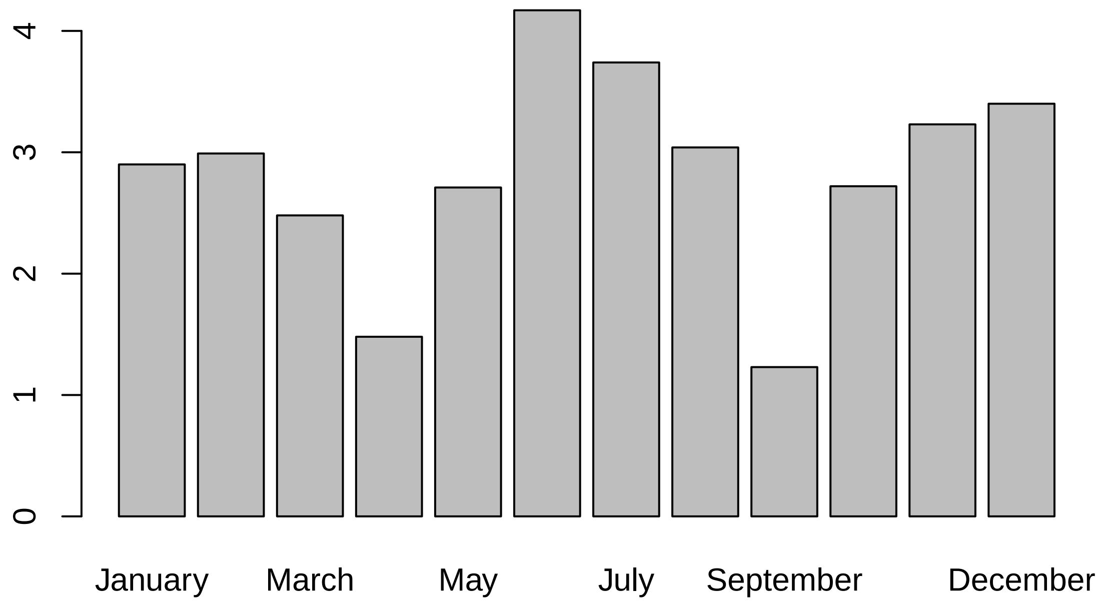
A bar plot

```
mydata <- read.csv("PrimerExamples/One-sample t-test.csv")
mydata
```

```
##        Month Failure.rate
## 1    January         2.90
## 2   February         2.99
## 3      March         2.48
## 4      April         1.48
## 5        May         2.71
## 6       June         4.17
## 7       July         3.74
## 8     August         3.04
## 9  September         1.23
## 10   October         2.72
## 11  November         3.23
## 12  December         3.40
```

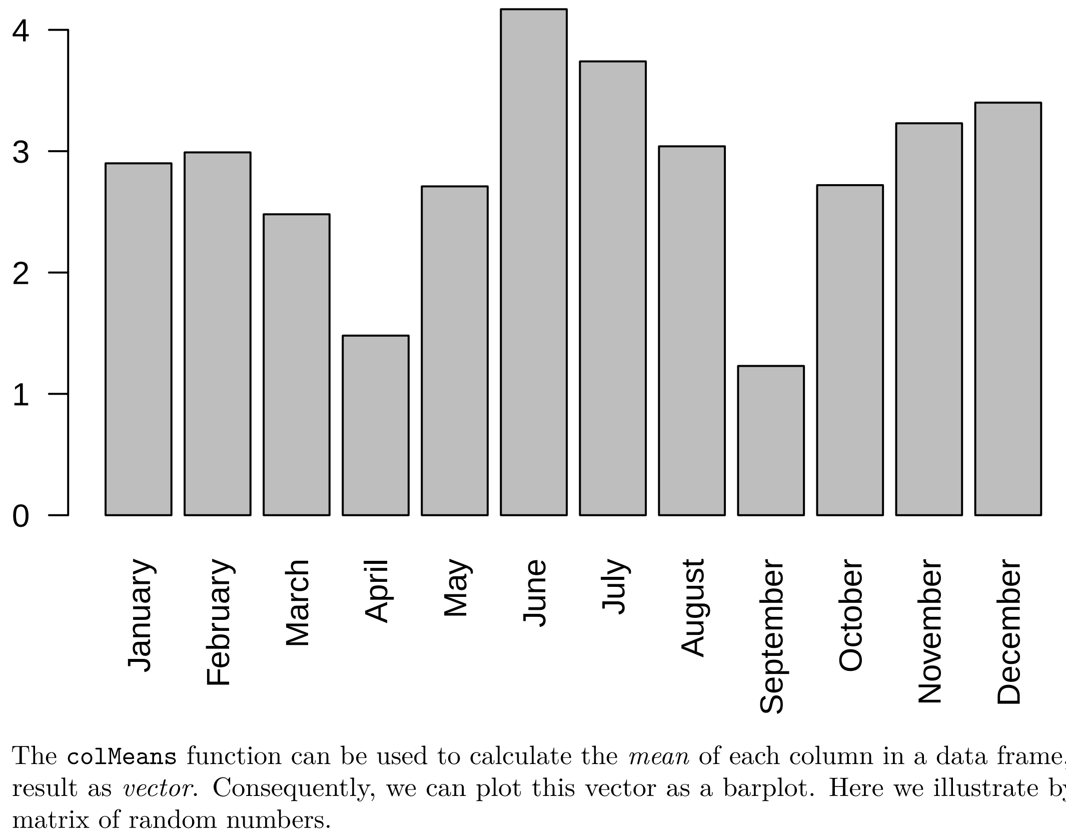
We can add a name underneath each bar by changing the `names.arg` argument.

```
barplot(mydata[,2],names.arg=mydata[,1])
```



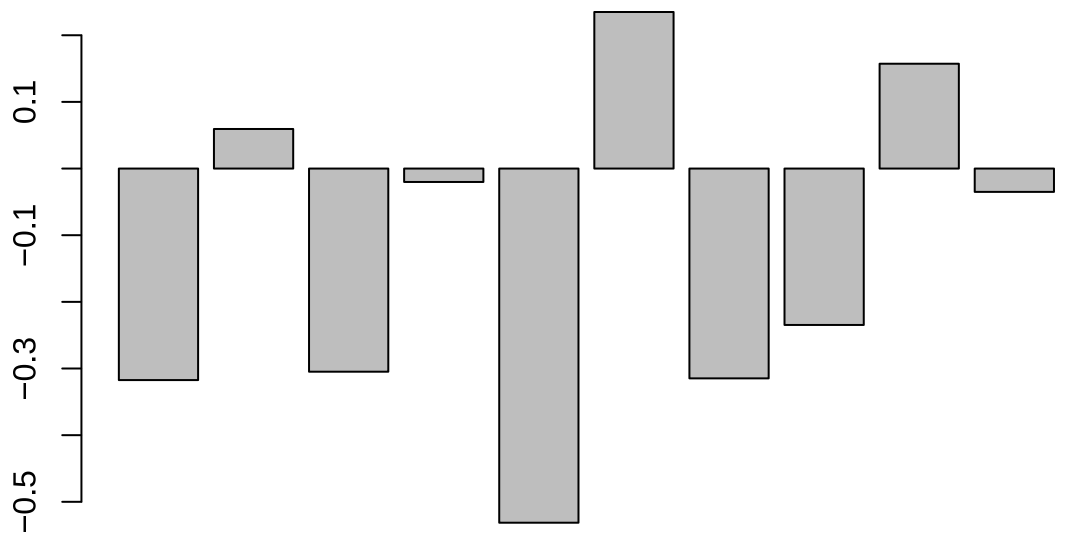
However, the default positioning of the labels is parallel to the axis and not all of the labels will fit in this space. We can change the orientation of the labels using the `las` argument. 0:= always parallel to the axis [default], 1: =always horizontal,2:=always perpendicular to the axis,3:=always vertical.

```
barplot(mydata[,2],names.arg=mydata[,1],las=2)
```



The `colMeans` function can be used to calculate the *mean* of each column in a data frame, and returns the result as *vector*. Consequently, we can plot this vector as a barplot. Here we illustrate by first creating a matrix of random numbers.

```
mydata <- matrix(rnorm(100),ncol=10)
means <- colMeans(mydata)
barplot(means)
```



The A.B.M dataset describes cases of Acute Bacerial Meningitis.

```
abm <- read.csv("data//ABM.csv")
head(abm)
```

```
##   casenum year month         age  race    sex dx priordx priorrx       wbc
## 1       1   78     1  4.00000000 black female  1       0       0  6.500000
## 2       2   78    12  1.00000000 black   male  1       0       0  3.700000
## 3       3   78     3  0.79999995 black   male  0       1       1         .
## 4       4   78     8 54.00000000 black   male  6       2       0  7.500000
## 5       5    .     .           .     .      .  .       .       .         .
## 6       6    .     .           .     .      .  .       .       .         .
##   pmn bands compns daysrx offrx lptodc lpgap morelabs bloodgl   gl   pr
## 1  50     4      0     10     0      0     .        .     165  3.0  304
## 2  62     5      0     10     2      0     .        0     150 92.0    .
## 3   .     .      6     10     1      1     .        .     183 36.0    .
## 4  73     7      6     10     .      .     .        0       . 52.0   43
## 5   .     .      .      .     .      .     .        .       .    .    .
## 6   .     .      .      .     .      .     .        .       .    .    .
##    reds whites polys lymphs monos others gram culture cie bloodclt
## 1   440   4000   100      0     0      0    4       1   1        1
## 2   450   5490    97      3     0      0    5       1   .        1
## 3     0   4500   100      0     0      0    0       0   0        .
## 4    27      0     .      .     .      .    0       6   .        6
## 5     .      .     .      .     .      .    .       .   .        .
## 6     .      .     .      .     .      .    .       .   .        .
##   bloodgl2 gl2 pr2 reds2 whites2 polys2 lymphs2 monos2 others2    sumbands
## 1        .  58  46     1      47      0     100      0       0   2.0000000
## 2        .  60   .     .       .      .       .      .       .   3.0990000
## 3        .  52  51   335     230      0     100      0       0           .
## 4        .   .   .     .       .      .       .      .       .   5.1089993
## 5        .   .   .     .       .      .       .      .       .           .
## 6         .   .   .     .       .      .       .      .       .          .
##      subset abm
## 1      test   1
## 2 training   1
## 3      test   .
## 4 training   1
## 5      test   0
## 6 training   0
```

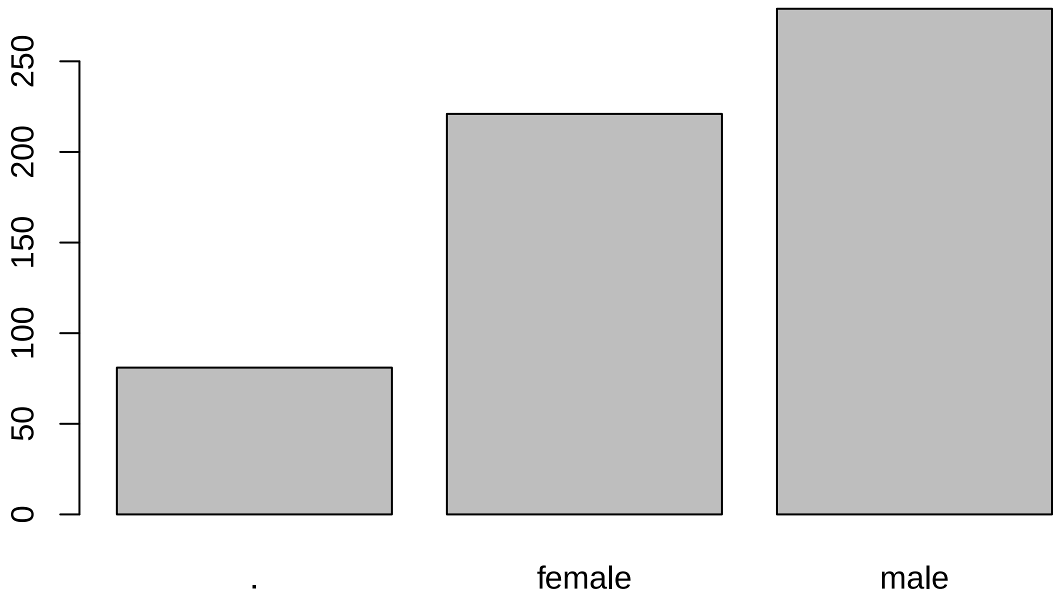
```r
dim(abm)
```

```
## [1] 581  43
```

```r
table(abm$sex)
```

```
##
##       . female   male
##      81    221    279
```

```r
barplot(table(abm$sex))
```

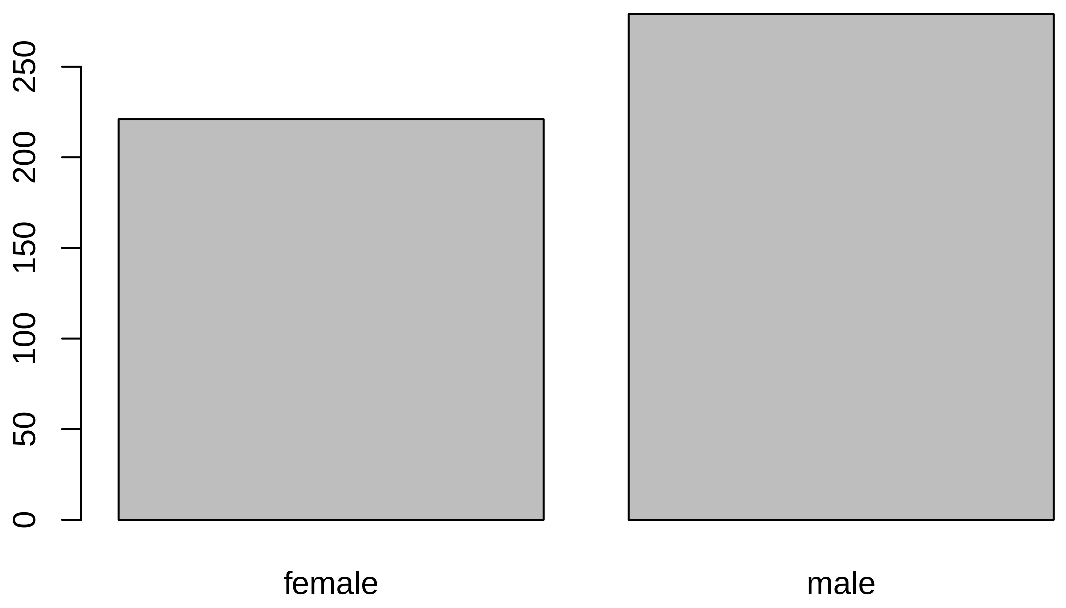


We can notice from the table and plot that a third gender category has been included; `.`. In this dataset, the `.` character is being used to represent a missing value. We can change this behaviour at the point of reading the data by changing the `na.strings` argument to `read.csv`.

```r
abm <- read.csv("data//ABM.csv",na.strings=".")
table(abm$sex)
```

```
##
## female   male
##    221    279
```

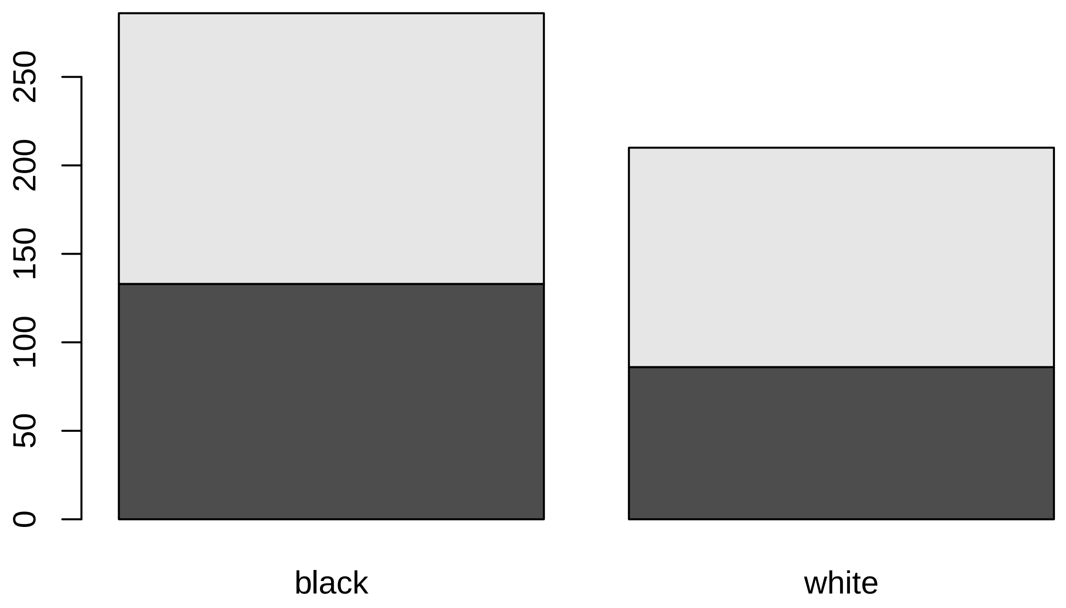
```
barplot(table(abm$sex))
```



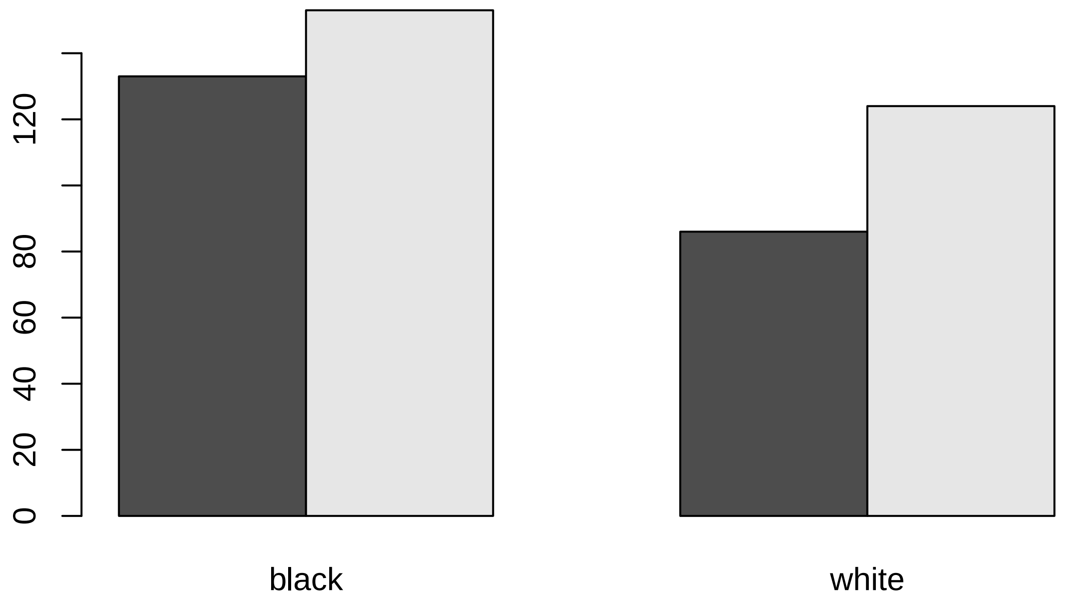
```
table(abm$sex,abm$race)
```

```
##
##          black white
##   female   133    86
##   male     153   124
```

```
barplot(table(abm$sex,abm$race))
```

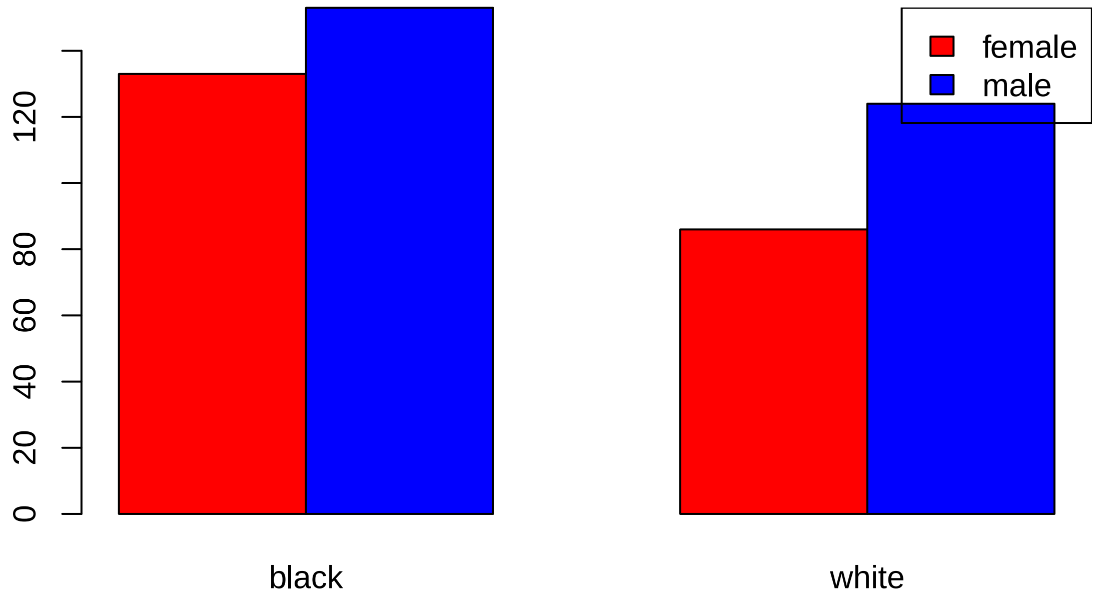


```
barplot(table(abm$sex,abm$race),beside=TRUE)
```



The plot can be improved by adding a legend and some colour.

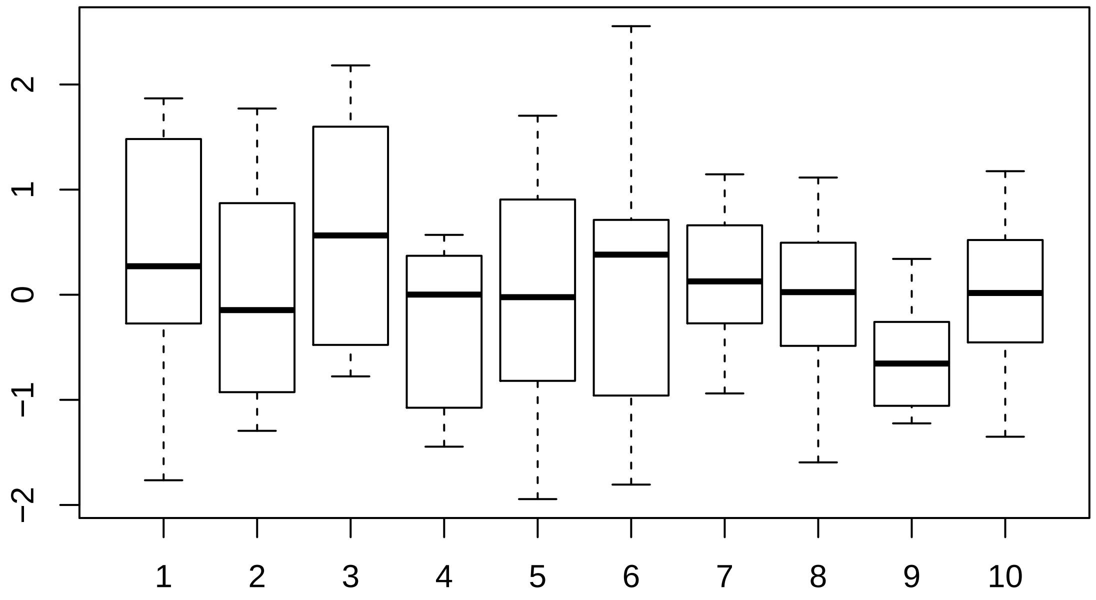
```
barplot(table(abm$sex,abm$race),beside=TRUE,col=c("red","blue"))
legend("topright", fill=c("red","blue"),legend=c("female","male"))
```



A boxplot

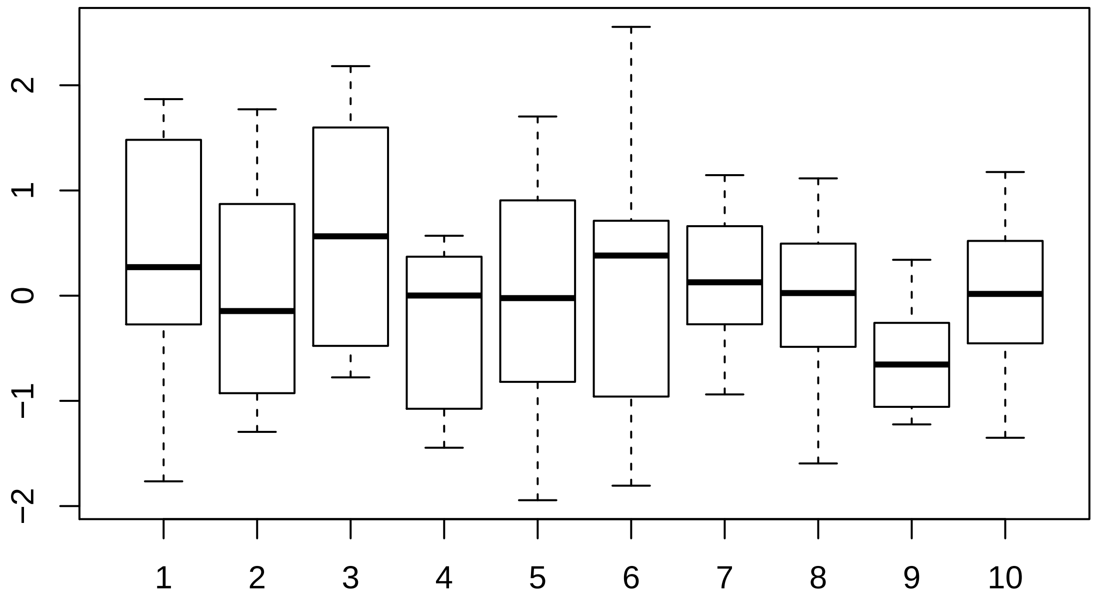
The boxplot is a convenient way of comparing the distributions of numeric data. If given a numeric matrix, or data frame, it will construct a box from the quartiles of the data . The whiskers are plotted at 1.5 times the inter-quartile range. Outliers are defined to be more than 1.5 times away from the box and plotted as circle.

```
mydata <- matrix(rnorm(100),ncol=10)
boxplot(mydata)
```



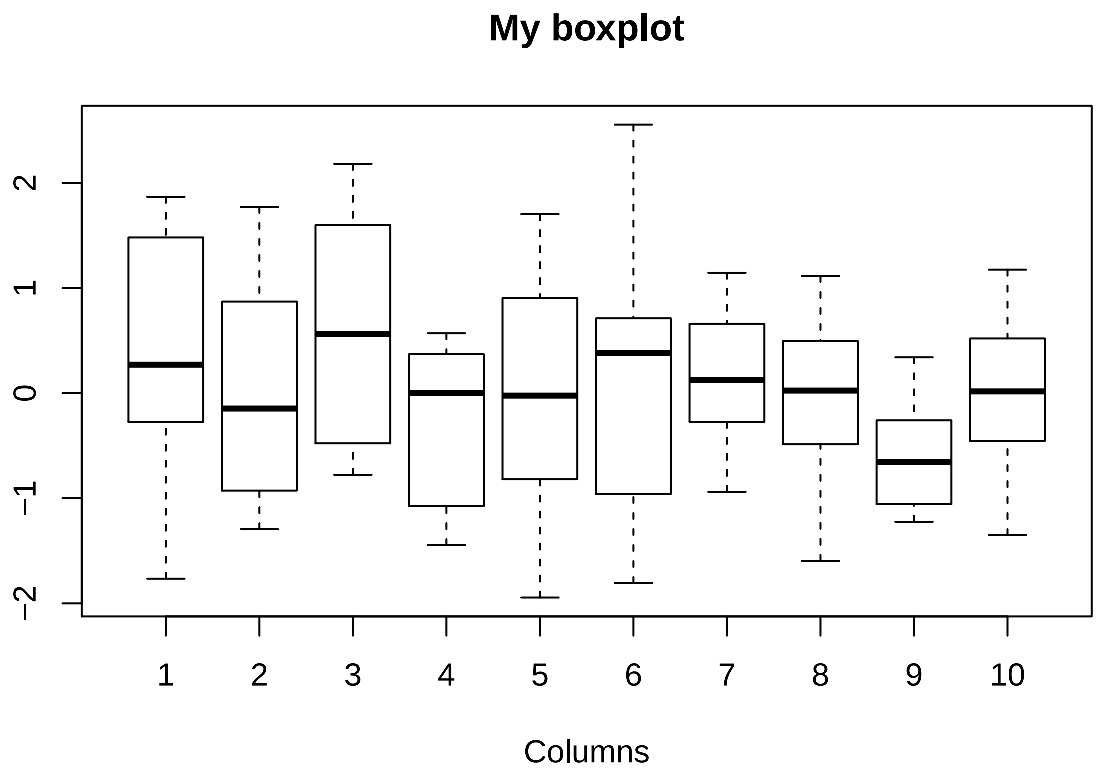
Outliers can be omitted from the plotting

```
boxplot(mydata,outline = FALSE)
```



The title and axis labels can be modified in the usual manner.

```
boxplot(mydata,outline = FALSE,main="My boxplot", xlab="Columns")
```



A boxplot can also be generated if our data are in *long* rather than *wide* format. One such dataset is the `mtcars` dataset, which is one of the built-in datasets in R.

One of the variables, `cyl` can only take certain values and is therefore a *categorical* variable. We can therefore use it to split our observations into different groups. The `mpg` variable (miles-per-gallon) is continuous. By treating the data in an appropriate way, we can get the distribution of `mpg` values for each distinct value of `cyl` in the data. Such an aggregation is done by specifying a *formula* using the `~` operator; the expression `mtcars$mpg~mtcars$cyl` does just this.

```
data(mtcars)
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
length(mtcars$mpg)
```

```
## [1] 32
```

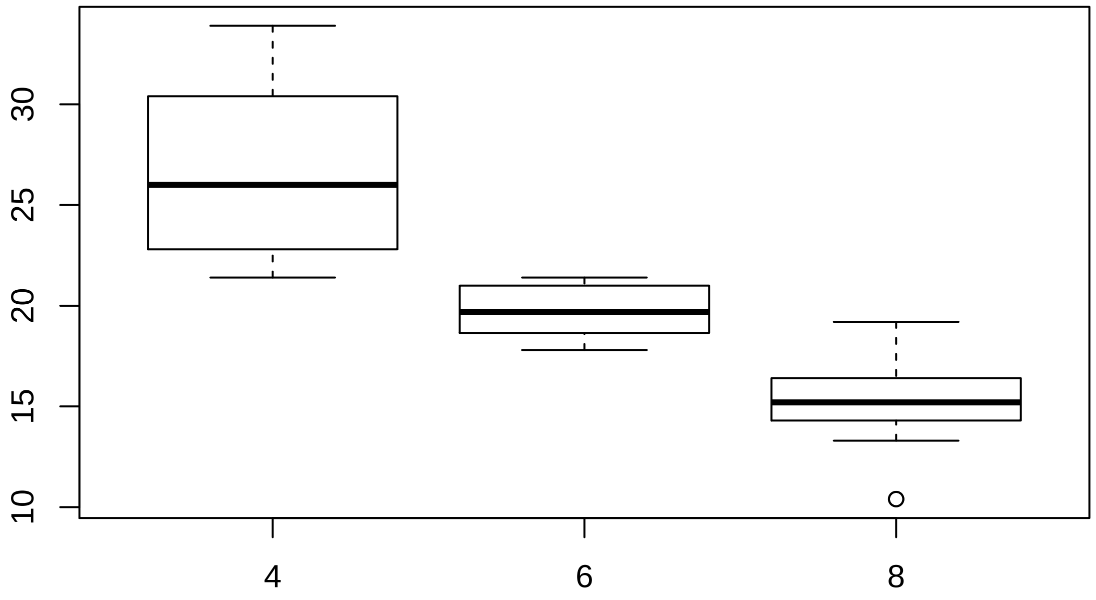
```
length(mtcars$cyl)
```

```
## [1] 32
```

```
table(mtcars$cyl)
```

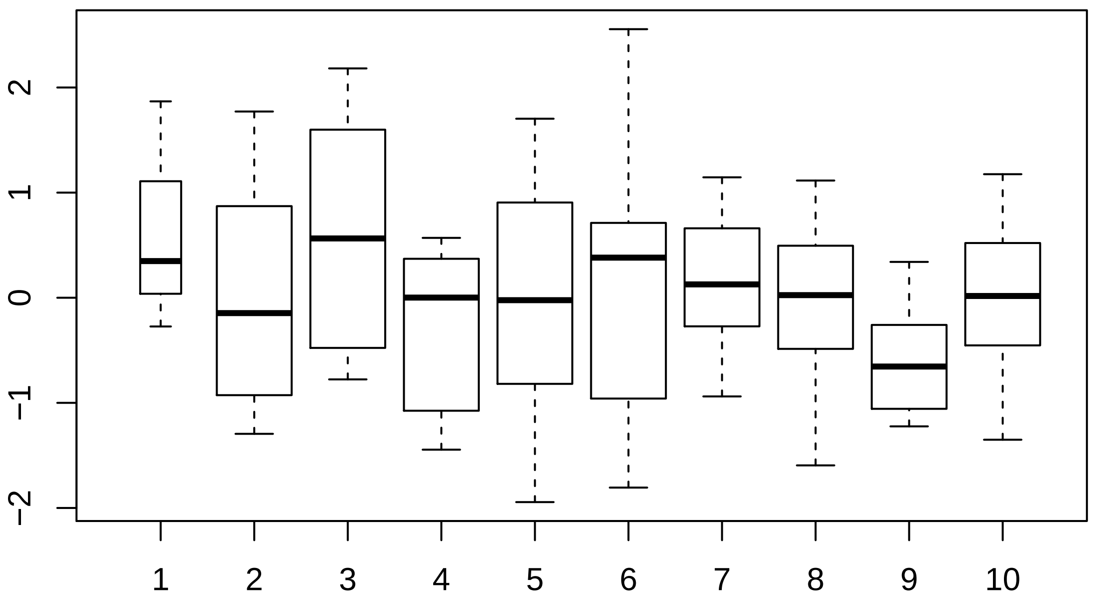
```
##
##  4  6  8
## 11  7 14
```

```
boxplot(mtcars$mpg~mtcars$cyl)
```



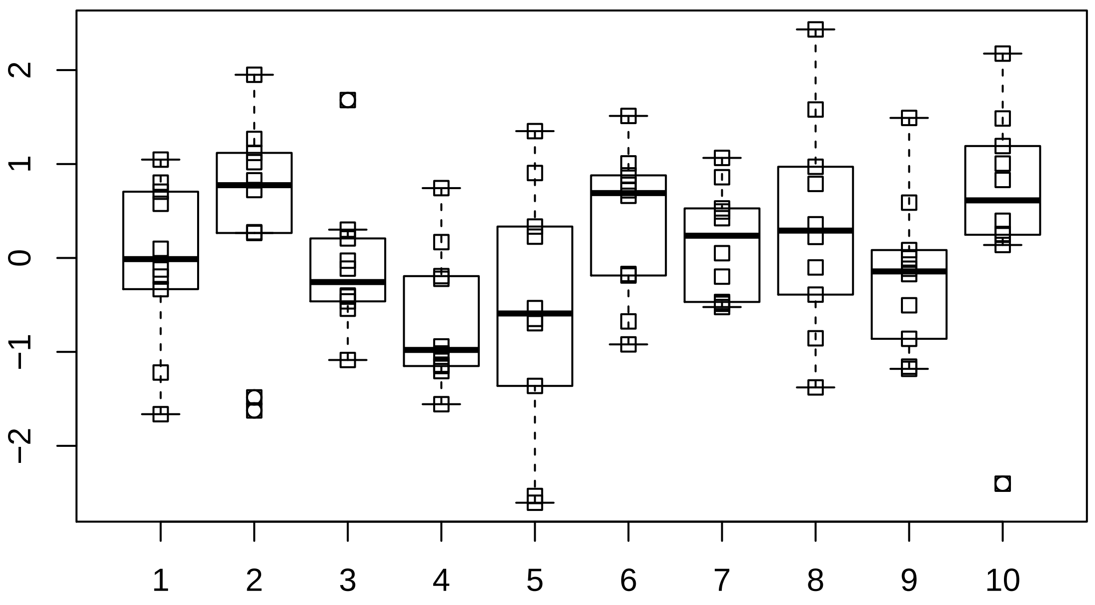
A boxplot can sometimes be misleading, as it does not provide any information about how many observations are in each group. One thing we can do is adjust the widths of the bars to reflect the differing number of observations.

```
mydata[1:7,1] <- NA
boxplot(mydata, varwidth = TRUE)
```



Dotchart / stripchart

```r
mydata <- matrix(rnorm(100),ncol=10)
boxplot(mydata)
stripchart(as.data.frame(mydata),add=TRUE,vertical=TRUE)
```



dotchart(mydata,vertical=TRUE)

Survival Curve

Growth curve

Replicating GraphPad Prism plots in R

Exporting a plot

R is able to save graphics in a variety of formats. The easiest way of saving a plot is to use the *Export* tab in the plot window in RStudio. You will then have the option of saving the plot as a *.pdf* or a variety of other file formats such as *.jpeg*, *.png*, *.tif.*

N.B. *.pdf* plots can be futher altered in a graphics program such as *Photoshop*, whereas *.jpeg*, *.png* can be more easily inserted into presentations. Journals often require high-resolution images in *.tiff* or *.eps* format.

The lines of code required to save a plot to a file are almost identical to producing the plot within RStudio; the only difference is that you need to send the the plot to an alternative *graphics device* other than RStudio, and close the device afterwards.

There are various functions can be used to create an alternative graphics device. These include `pdf`, `jpeg`, `png` and have to be called before the code that creates the plot. You have to specify a file name that the plot will be written to. However, the file does not have to exist prior to creating the plot.

In the following example, we will save a boxplot as a *.png* file. We will therefore use the `png` function to write the plot to a file. If you run the code, you will notice that the plot is no longer displayed in RStudio. This is because the plot has been created in an alternative graphics *device*. We therefore have to use the `dev.off` function to close this device and allow us to open the file.

```
mydata <- read.csv("PrimerExamples/Paired two-sample t-test.csv")
file.exists("myboxplot.png")
```

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

```
list.files(pattern=".png")
```

```
## [1] "mycoolplot.png"
```

```
png("myboxplot.png")
boxplot(mydata)
dev.off()
```

```
## pdf
##   2
```

You should notice that a file `myboxplot.png` has been created in your working directory. We can verify this using the `file.exists` and `list.files` functions.

```
file.exists("myboxplot.png")
```

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

```
list.files(pattern=".png")
```

```
## [1] "myboxplot.png"  "mycoolplot.png"
```

To create a *.jpeg* instead of a *.png*, we simply use `jpeg` instead of `png`.

```
jpeg("myboxplot.jpeg")
boxplot(mydata)
dev.off()
```

```
## pdf
##   2
```

You can specify the height and width of the plot by adjusting the `height` and `width` arguments. This can be done in the same way for the `jpeg` and `png` functions. The values for `height` and `width` should be the required number of pixels.

```
jpeg("myboxplot2.jpeg",width=800,height=500)
boxplot(mydata)
dev.off()
```

```
## pdf
##   2
```

.pdf documents as created in much the same way using the `pdf` function. However, you should note that the dimensions of the plot are specified in *inches* rather than *pixels*.

```
pdf("myboxplot.pdf",width=8,height=4)
boxplot(mydata)
dev.off()
```

```
## pdf
##   2
```

An important different between *.jpeg*, *.png* and *.pdf* is that *.pdf* documents can have multiple pages. In the following example, we have multiple lines of code between the `pdf` and `dev.off` lines. Each plot is saved onto a separate page in the pdf document.

```
pdf("myplots.pdf")
hist(mydata[,1])
hist(mydata[,2])
boxplot(mydata)
dev.off()
```

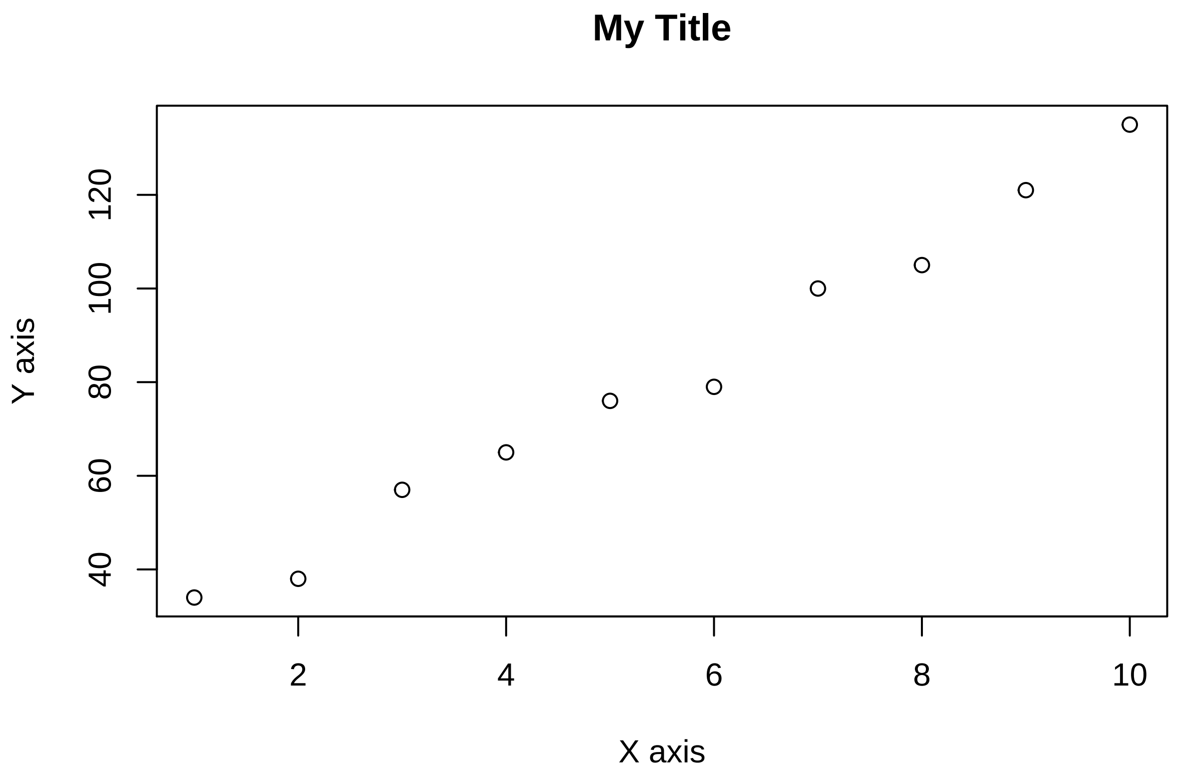
```
## pdf
##   2
```

Changing the plot appearance

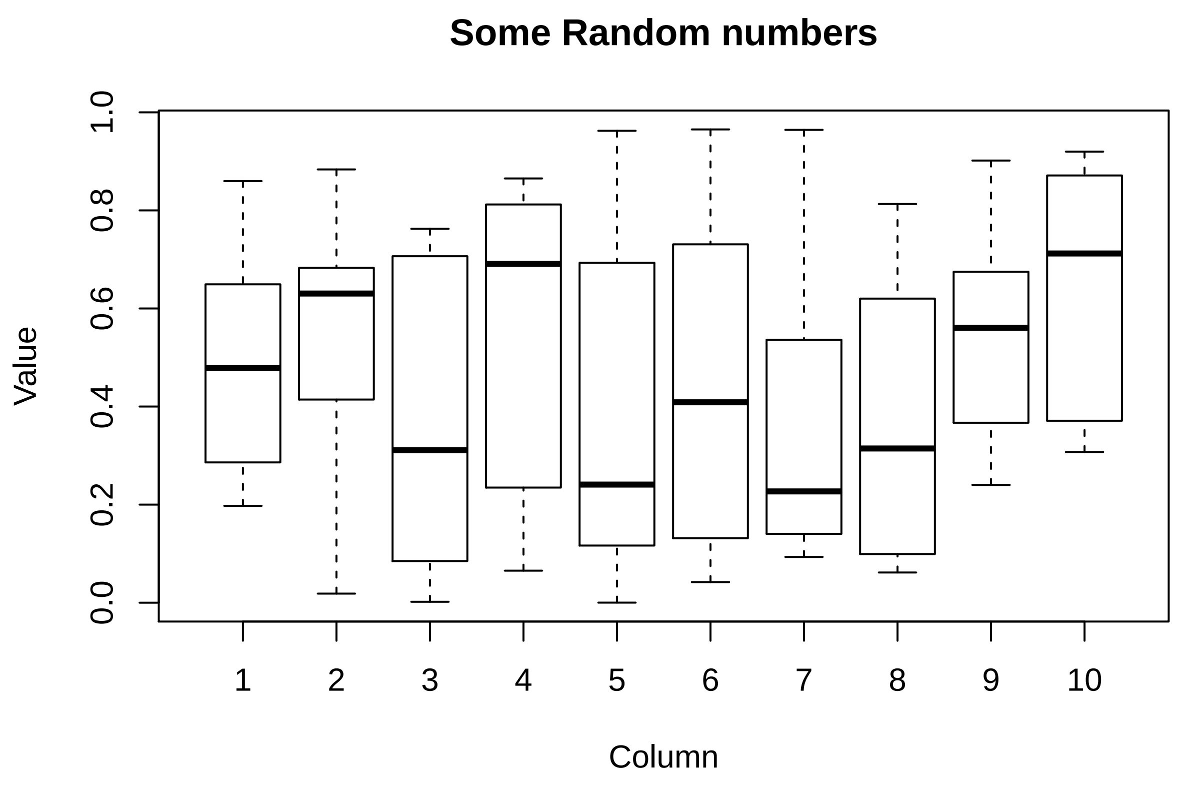
Titles and axis labels

It is always a good idea to use informative titles and axis labels on your plots. Unfortunately, the default plots produced by R try and *guess* labels from the data, which may not be that useful. The `plot` function has `main`, `ylab` and `xlab` arguments that can be used to specify these labels (which must be a character vector). Moreover, other plotting functions such as `boxplot`, `barplot` and `histogram` also use the same argument names.

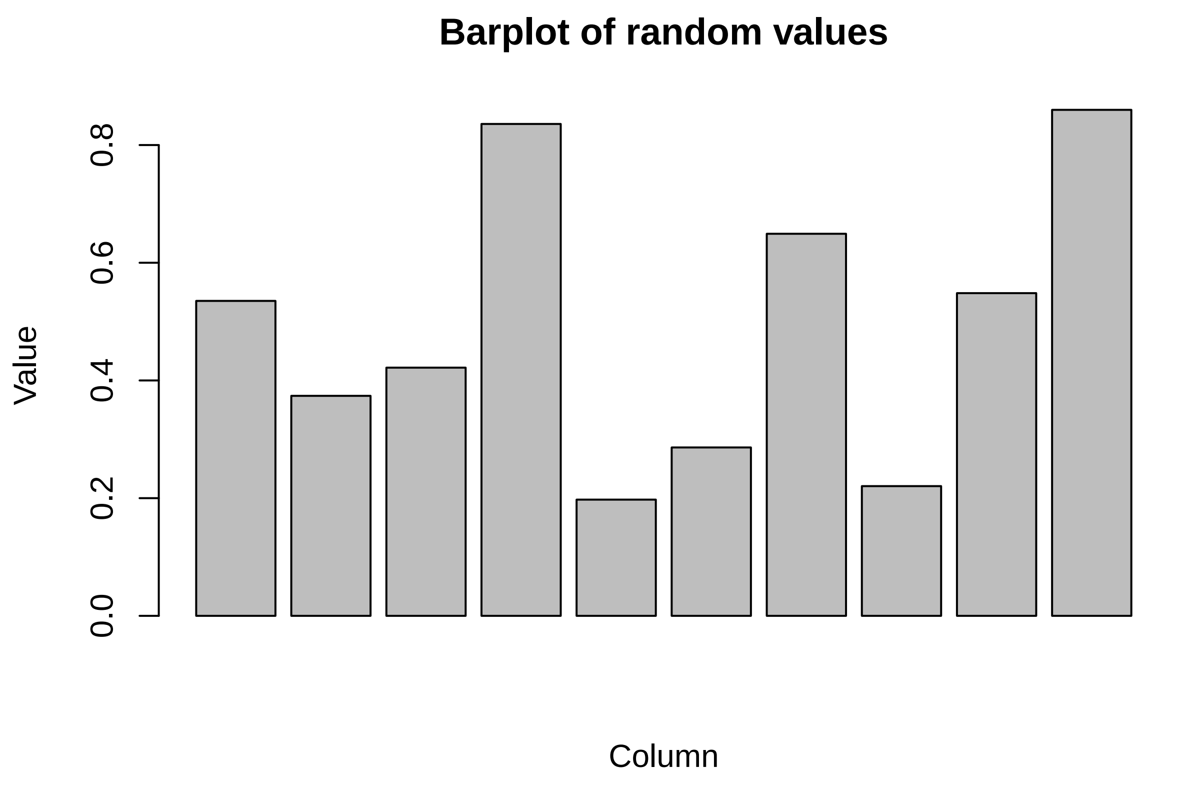
```r
mydata <- read.delim("PrimerExamples/Linear regression.txt")
plot(mydata[,1],mydata[,2],main="My Title", xlab="X axis", ylab="Y axis")
```



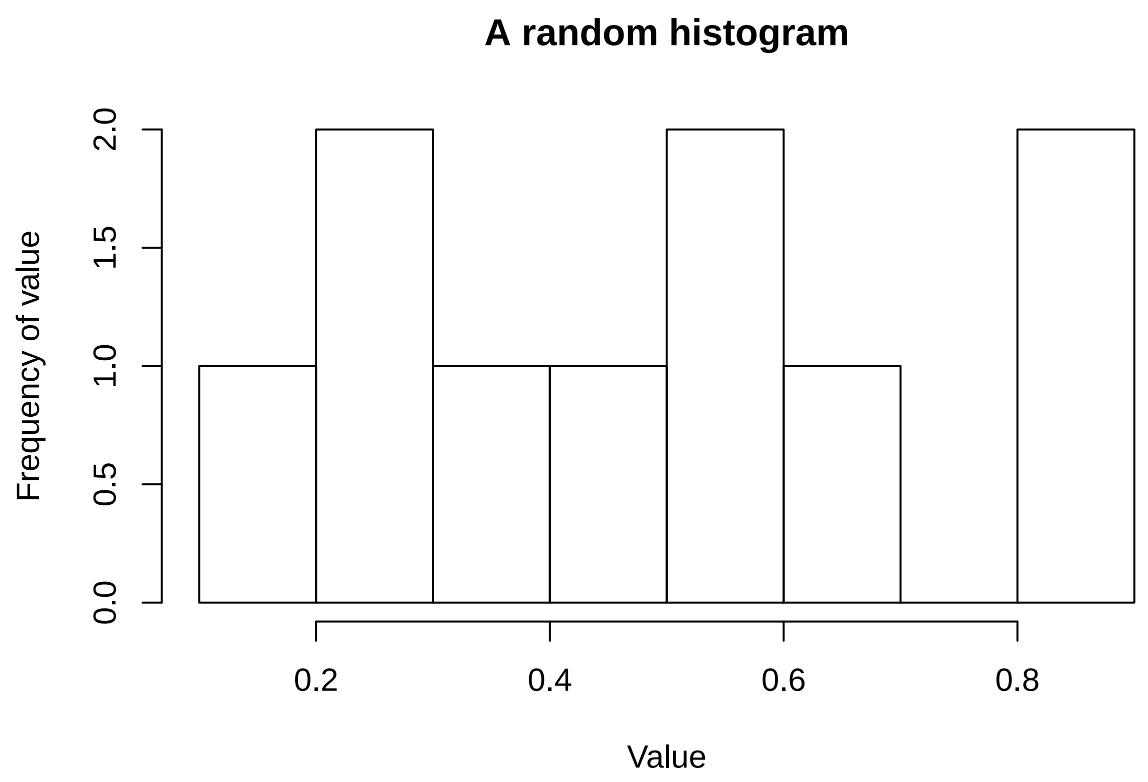
```r
randMat <- matrix(runif(100),ncol=10)
boxplot(randMat, main="Some Random numbers" ,xlab="Column",ylab="Value")
```



```r
barplot(randMat[,1], main="Barplot of random values",ylab="Value",xlab="Column")
```

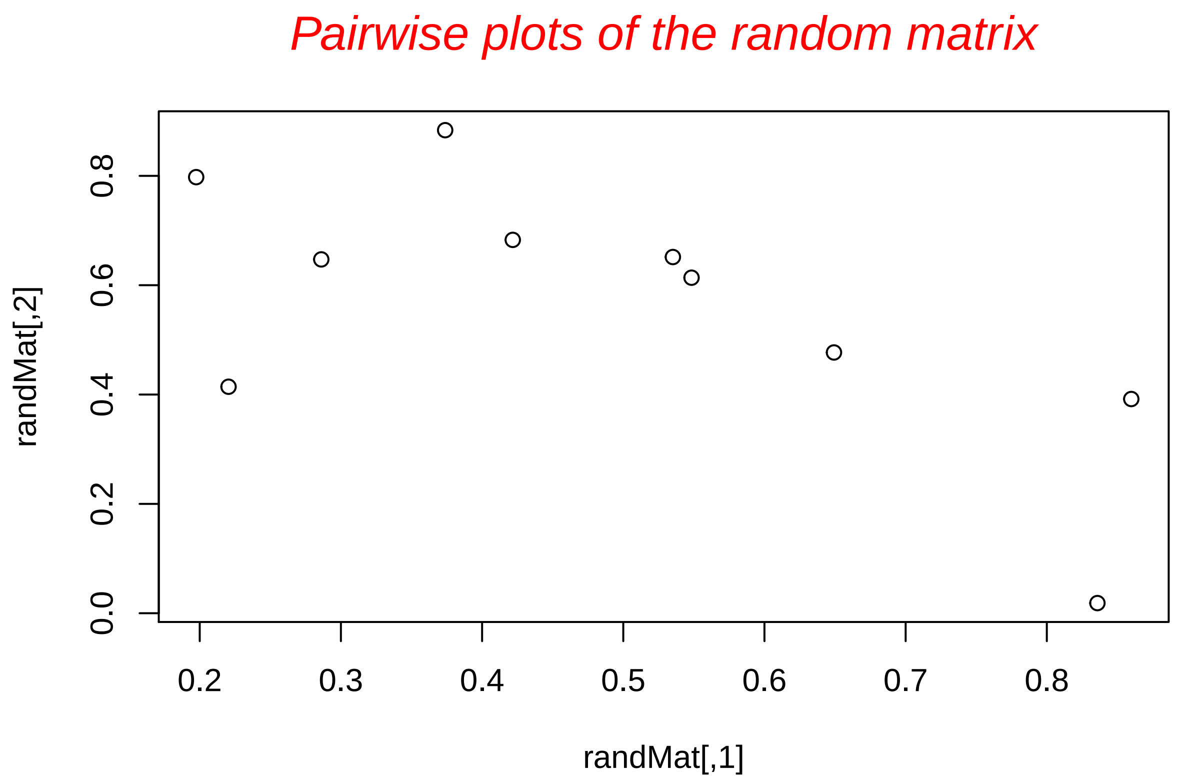


```r
hist(randMat[,1],main="A random histogram", xlab="Value", ylab="Frequency of value")
```



Functions such as `points`, `lines`, `text` and `abline` cannot modify the title or axis labels. We can set the plot title to be blank (the empty string `""`) at the time the plot is created, and add a title manually later-on using the `title` function. The following also demonstrates that the colour and font of the title can be modified.

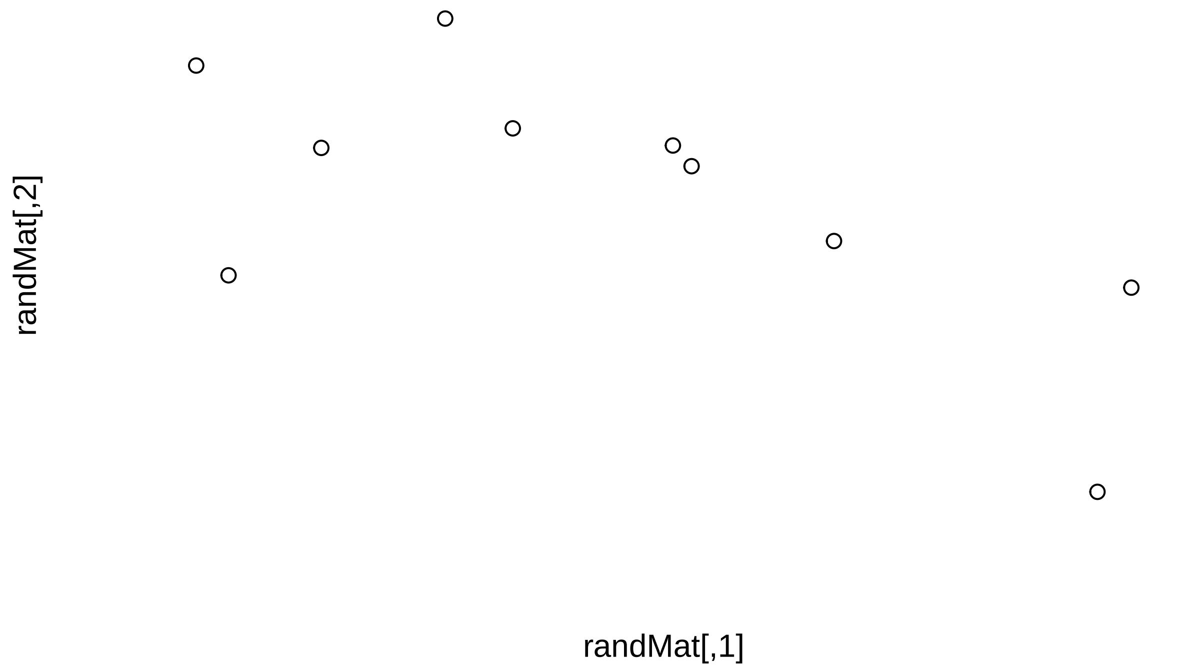
```
plot(randMat, main = "")
title(main = list("Pairwise plots of the random matrix", cex = 1.5,
                  col = "red", font = 3))
```



Adding custom axes

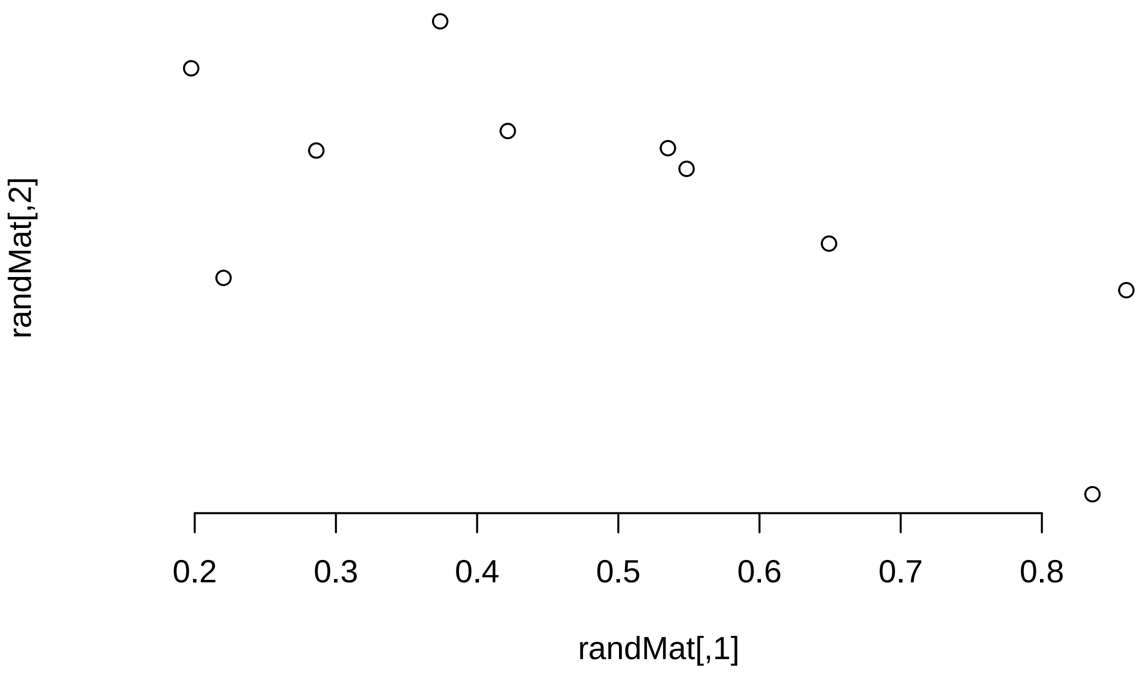
It is possible to create your own axes. First, you can to create a plot with the `axes=FALSE` argument. You will see in the resulting plot that no axes are plotted

```r
plot(randMat, axes=FALSE)
```

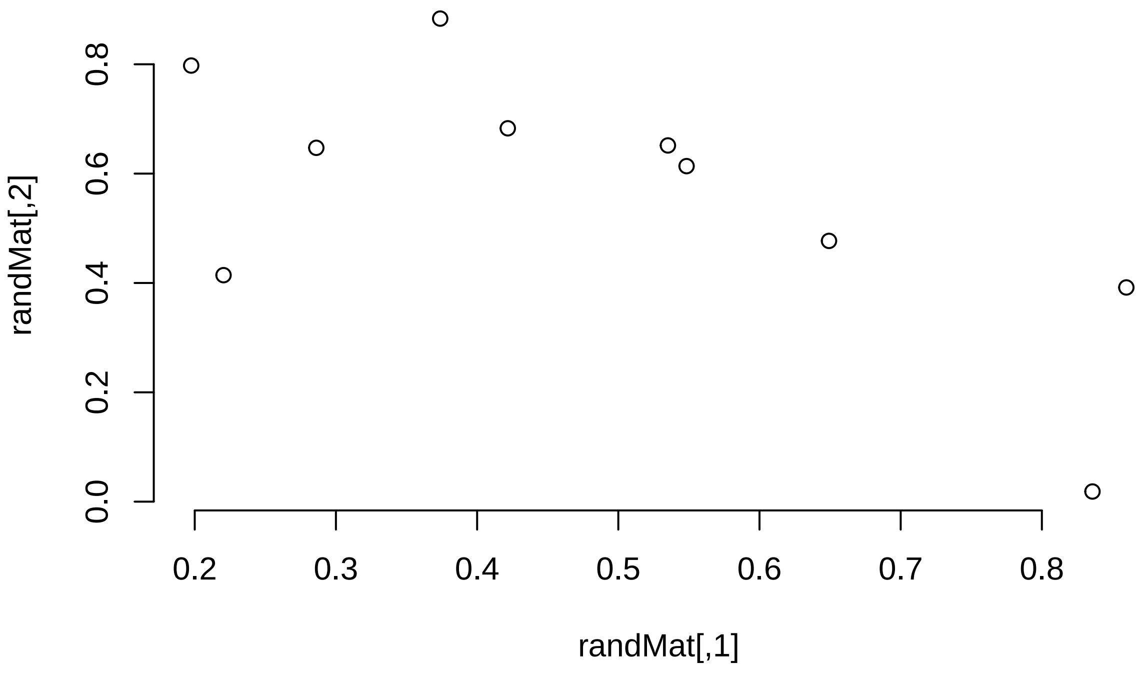


We can use the `axis` function to plot an axis at each side of the plot (1=below, 2=left, 3=above and 4=right). Each time the `axis` function is called, a new axis is overlaid on the current plot.

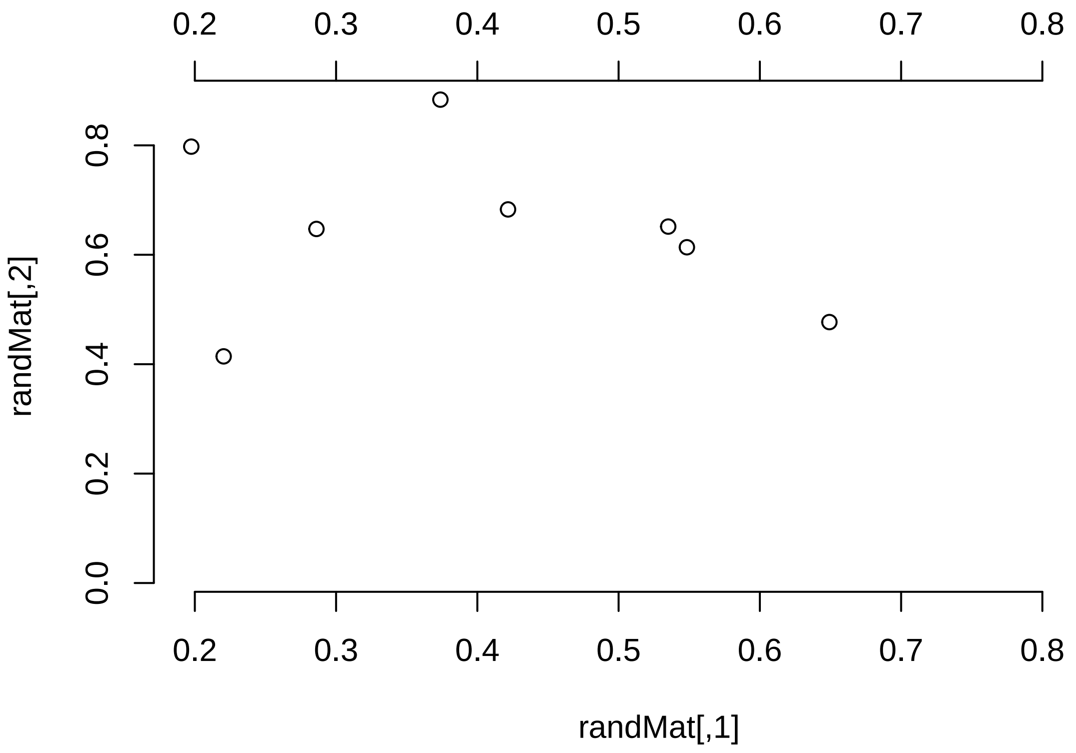
```r
plot(randMat, axes=FALSE)
axis(1)
```



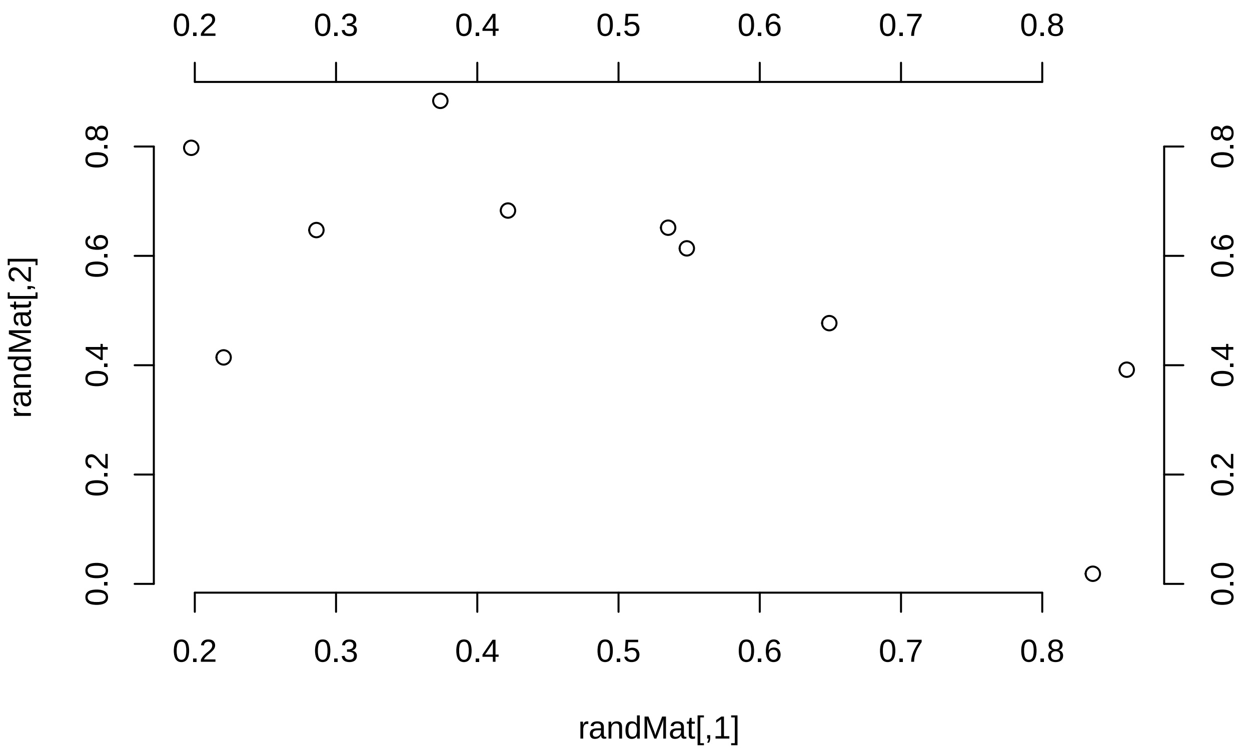
```r
plot(randMat, axes=FALSE)
axis(1)
axis(2)
```



```
plot(randMat, axes=FALSE)
axis(1)
axis(2)
axis(3)
```

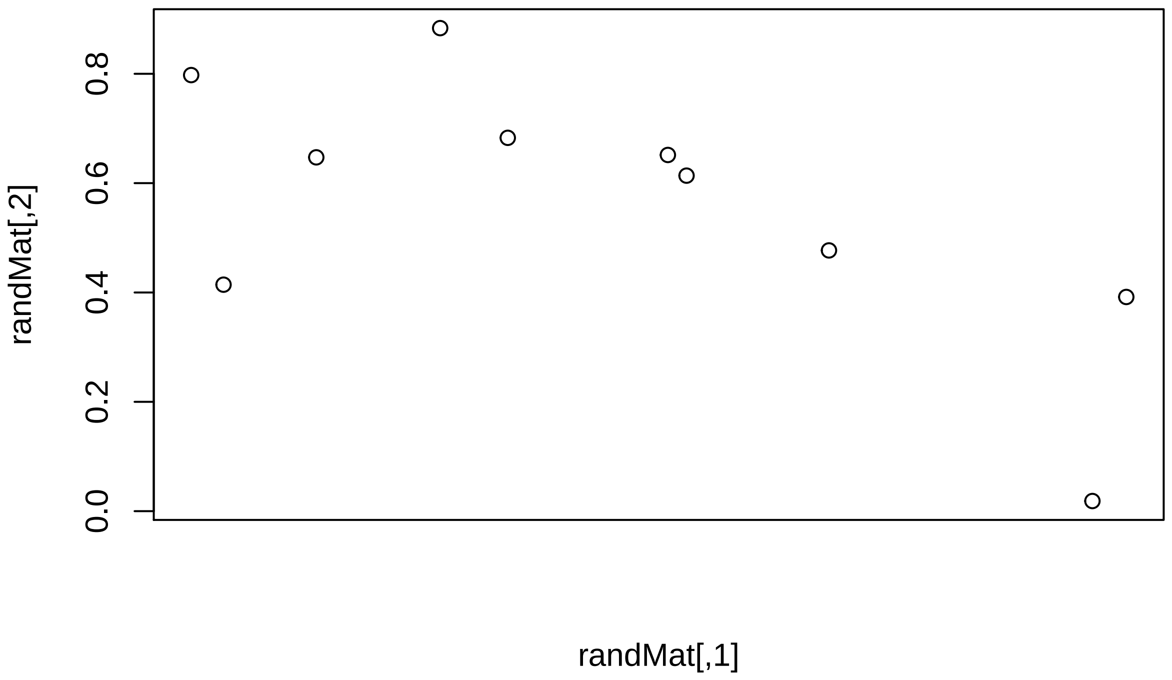


```
plot(randMat, axes=FALSE)
axis(1)
axis(2)
axis(3)
axis(4)
```



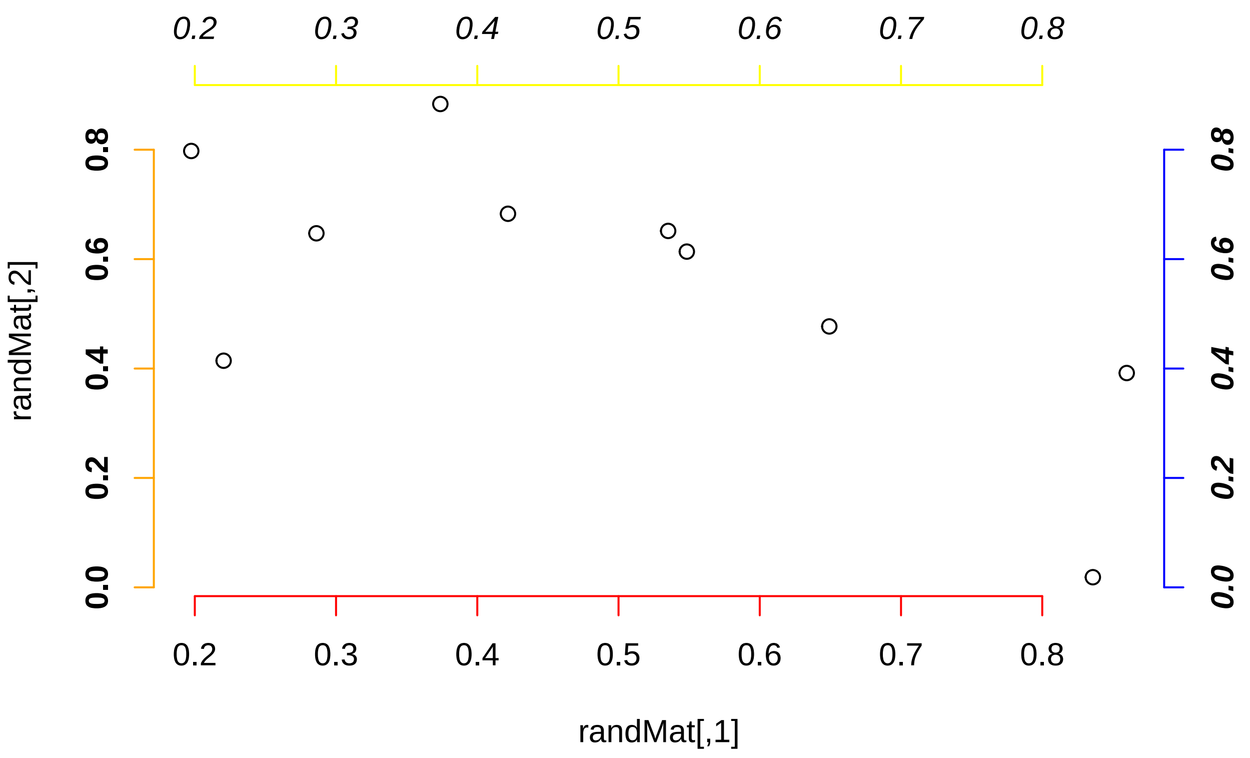
You don't need to specify an axis for each side of the plot. The `box` function will enclose the remaining sides without giving an axis.

```
plot(randMat, axes=FALSE)
axis(2)
box()
```



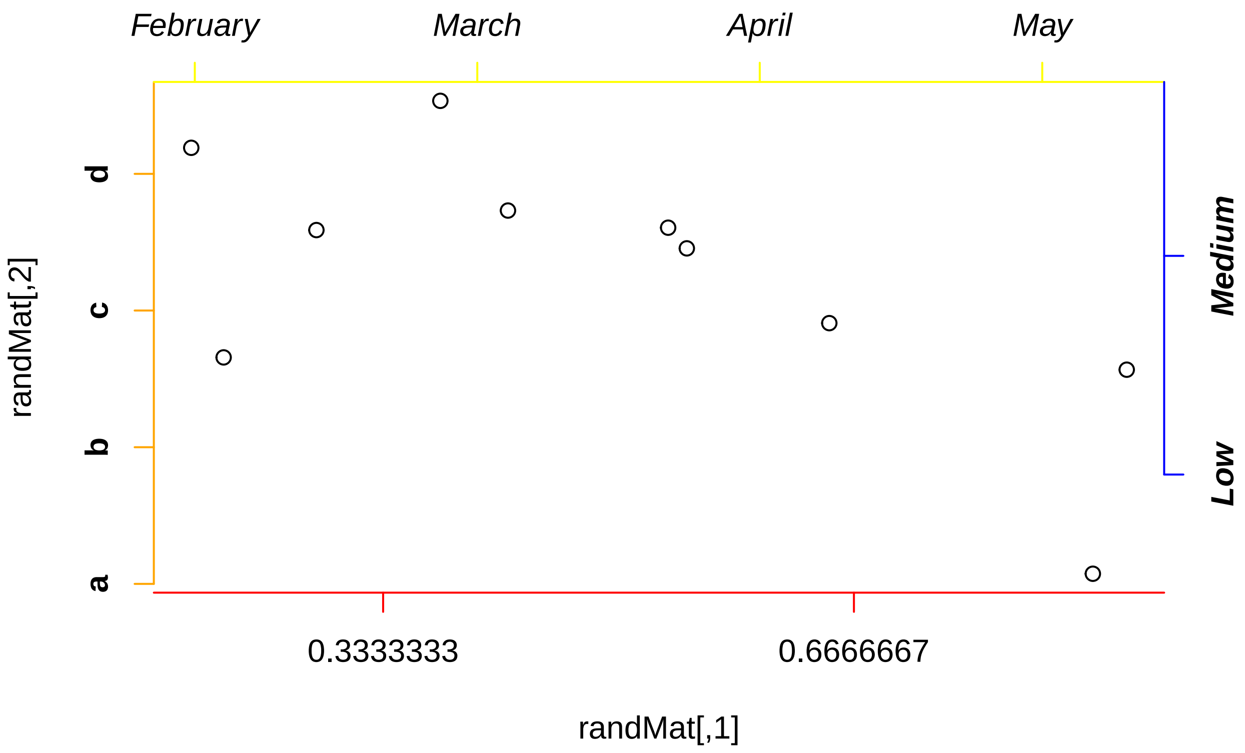
The font and colour of each axis can also be changed

```
plot(randMat, axes=FALSE)
axis(1,font=1,col="red")
axis(2,font=2,col="orange")
axis(3,font=3,col="yellow")
axis(4,font=4,col="blue")
```



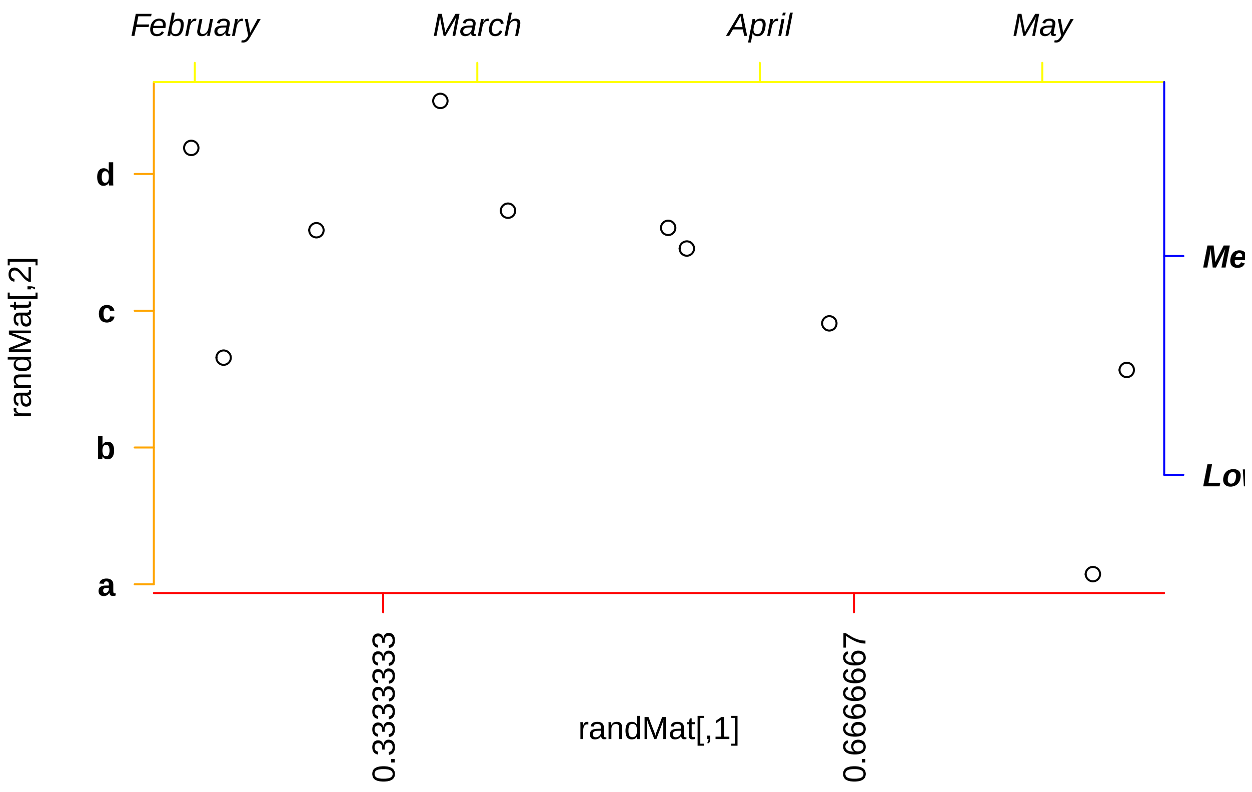
We can change the position and labels of the tick marks

```
plot(randMat, axes=FALSE)
axis(1,font=1,col="red",at=seq(0,1,length.out = 4))
axis(2,font=2,col="orange",at=seq(0,1,length.out = 5),labels=letters[1:5])
axis(3,font=3,col="yellow",at=seq(0,1,length.out = 6),labels=month.name[1:6])
axis(4,font=4,col="blue",at=seq(0.2,1,length.out = 3),labels=c("Low","Medium","High"))
```



The tick labels can be rotated so that they are perpendicular to the axis

```
plot(randMat, axes=FALSE)
axis(1,font=1,col="red",at=seq(0,1,length.out = 4),las=2)
axis(2,font=2,col="orange",at=seq(0,1,length.out = 5),labels=letters[1:5],las=2)
axis(3,font=3,col="yellow",at=seq(0,1,length.out = 6),labels=month.name[1:6])
axis(4,font=4,col="blue",at=seq(0.2,1,length.out = 3),labels=c("Low","Medium","High"),las=2)
```



Specifying Colours

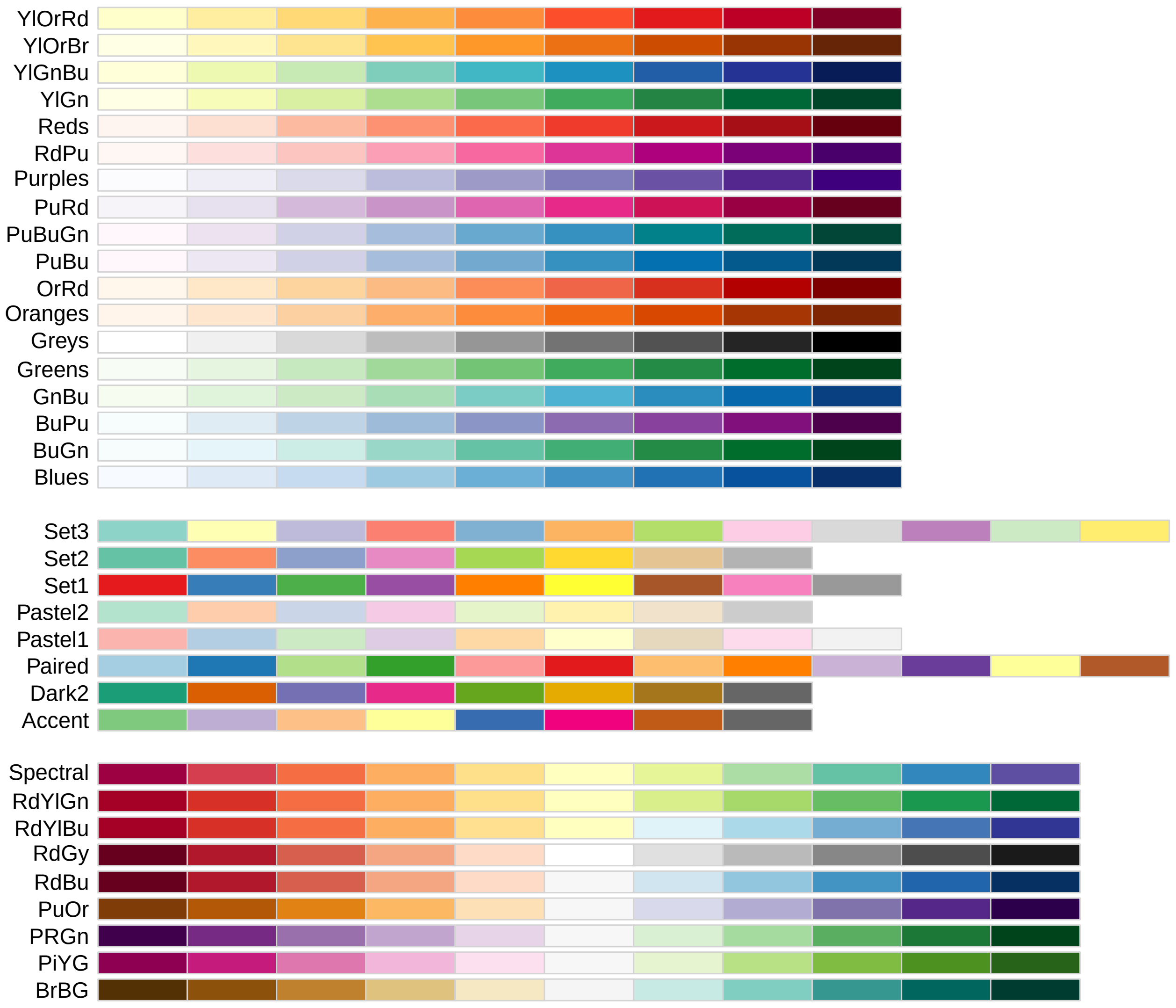
The simplest way of specifying a colour in R is to use a pre-defined character string. There are 657 preset values you can use, and their names can be printed to the screen using the `colors` function.

```r
colors()
```

There is a colour chart available online that can be used to display the colors alongside the names.

You should make sure that the colour scheme that you choose is suitable for those with colour-blindness. e.g. avoiding red / green colour schemes. One way of ensuring this is to use predfined pallettes in the `RColorBrewer` package.

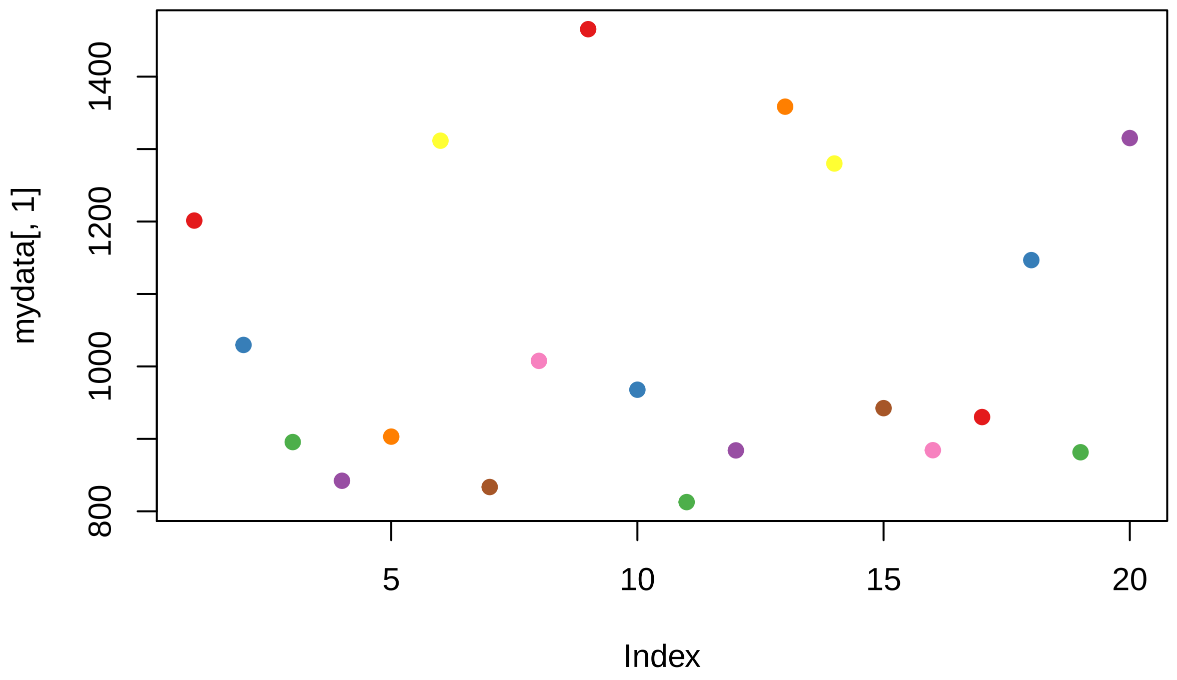
```r
library(RColorBrewer)
display.brewer.all()
```



You can pick one of these palette using the `brewer.pal()` function, where we specify first the number of distinct colours we want and then the name of the palette taken from the list in the image above.

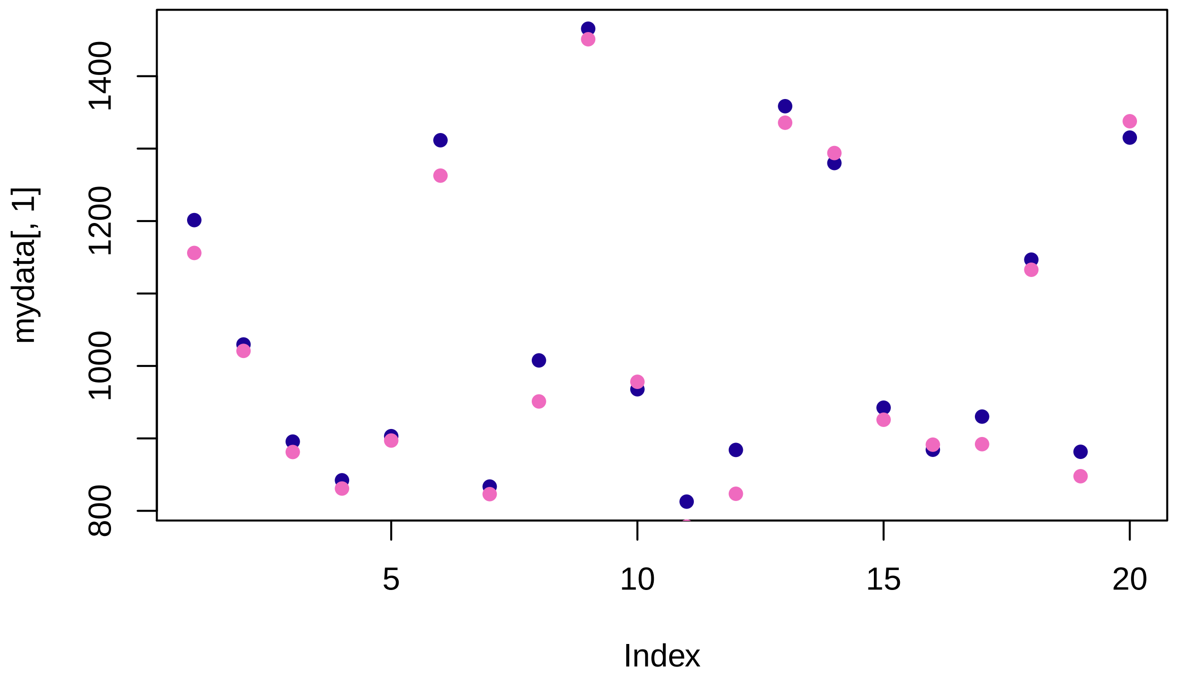
To use the colours in a figure, most plotting functions include the argument `col` to indicate the colours you wish to use.

```
mypal <- brewer.pal(8, "Set1")
mydata <- read.csv("PrimerExamples/Paired two-sample t-test.csv")
plot(mydata[,1], col = mypal, pch = 19)
```

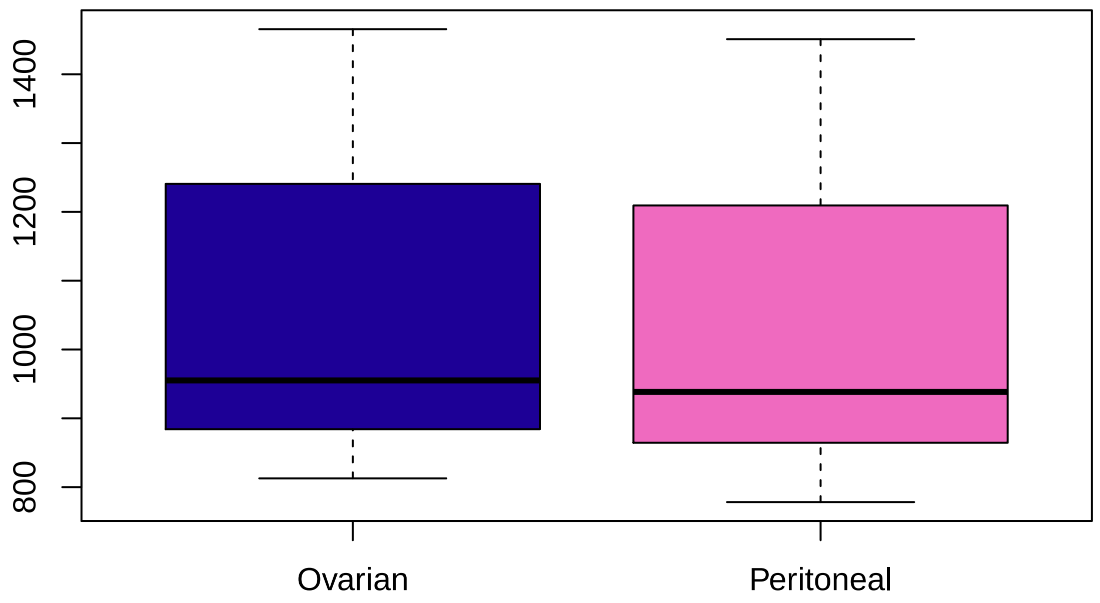


If you wish to use Cancer Research Uk or Cambridge University branding, you can use the colours defined in the `crukCIMisc` package. This package includes a function, `CRUKcol` which can be used to generate the Cancer Research UK blue or pink colours.

```r
library(crukCIMisc)
plot(mydata[,1], col=CRUKcol("Blue"),pch=16)
points(mydata[,2],col=CRUKcol("Pink"),pch=16)
```

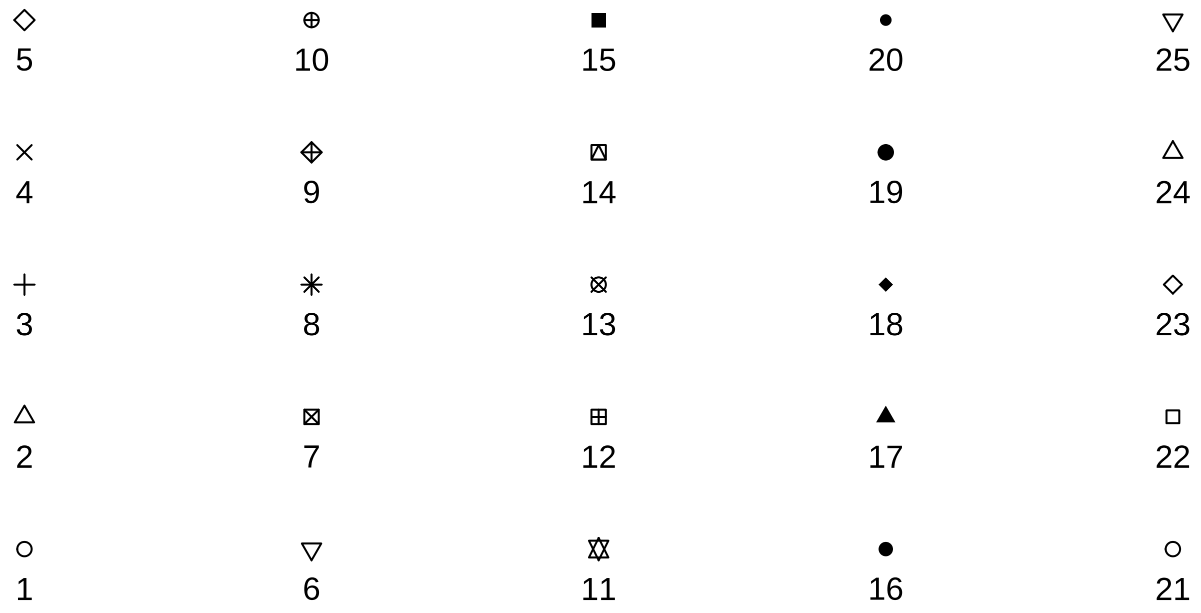


```r
boxplot(mydata, col=c(CRUKcol("Blue"),CRUKcol("Pink")))
```



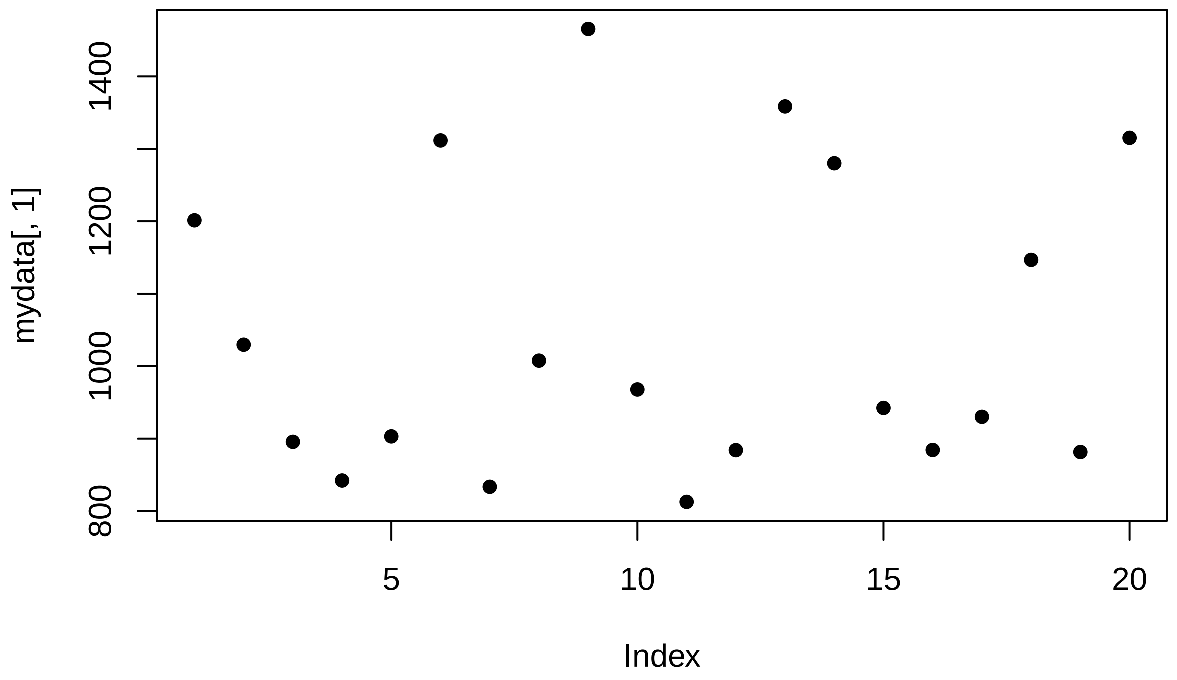
Plotting characters and their values

You may have noticed that in some of the example scatter plots above not only the colour of the points has been changing, but also the shape that is being drawn. The default is an open circle, but R has a variety of alternative shapes, which are listed in the diagram below.



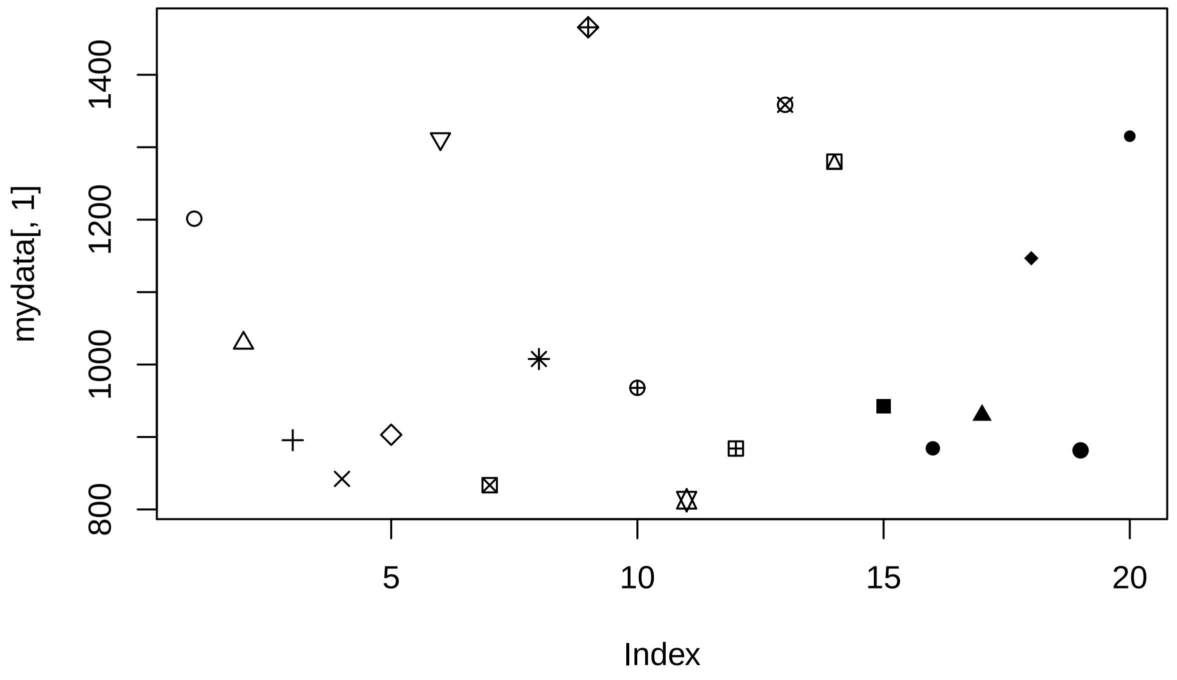
We can specify a new plotting character using the `pch` argument. For example `pch=16` uses filled rather than open circles.

```r
mydata <- read.csv("PrimerExamples/Paired two-sample t-test.csv")
plot(mydata[,1],pch=16)
```



Rather than being a single value, `pch` can be a vector of values that will be cycled through as each data point is plotted.

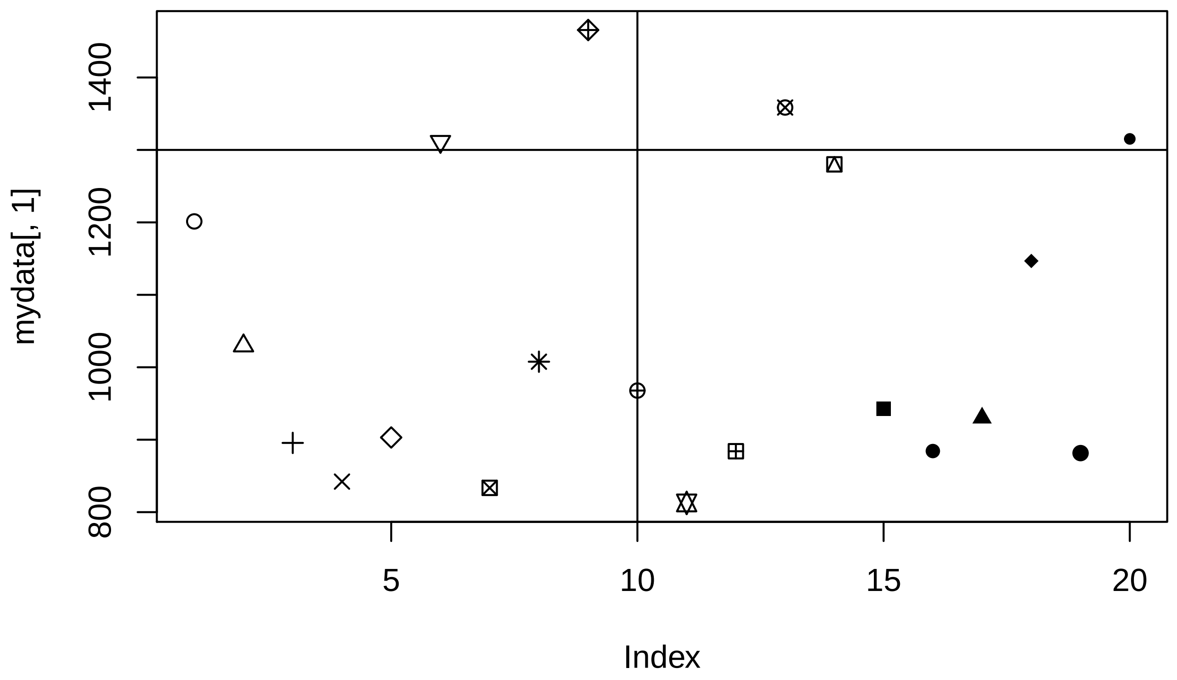
```r
plot(mydata[,1],pch=1:nrow(mydata))
```



Adding lines, points to a plot

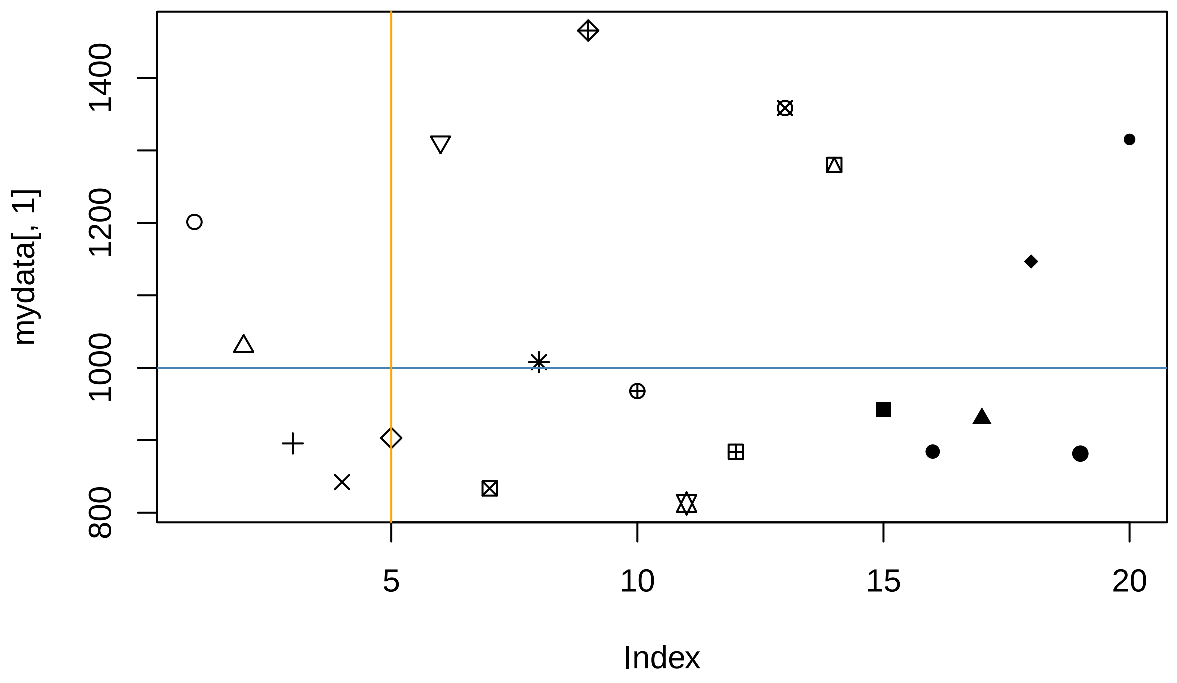
Once a plot has been created, we have various options to modify it. Firstly we can add vertical and horzontal lines to the plot using `abline` (read as *ab line*) with the `v` and `h` arguments respectively. Here we construct a scatter plot as before, and

```
plot(mydata[,1],pch=1:nrow(mydata))
abline(h=1300)
abline(v=10)
```



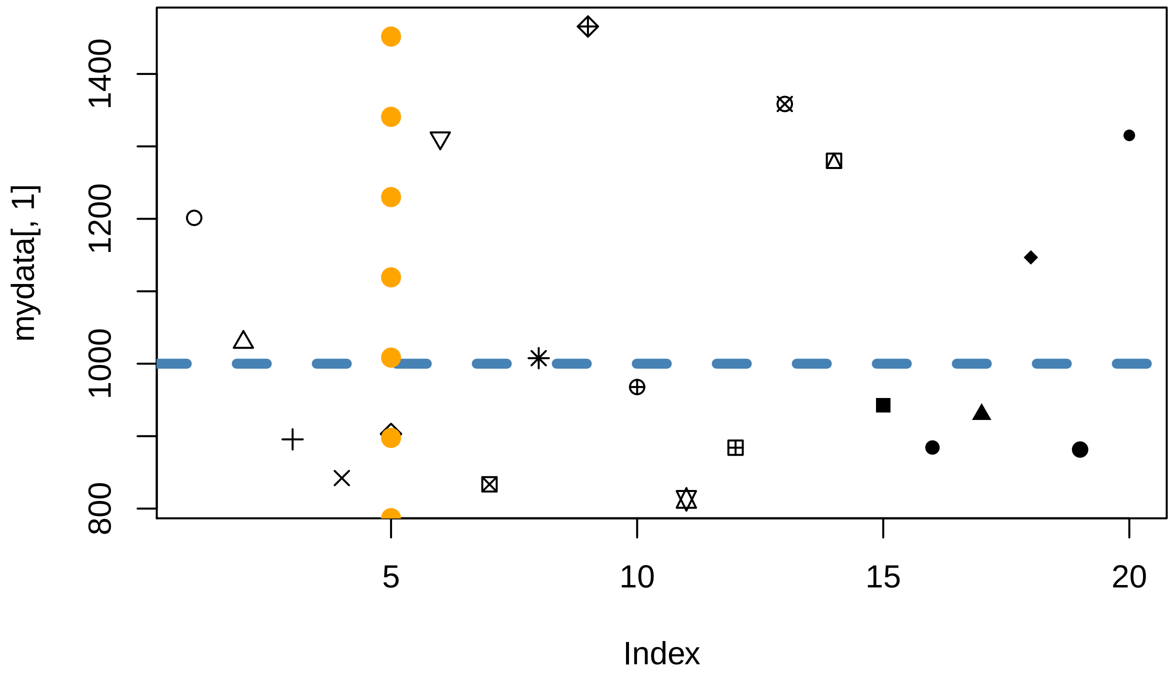
The appearance of the line can be modified, for example to be coloured differently.

```
plot(mydata[,1],pch=1:nrow(mydata))
abline(h=1000,col="steelblue")
abline(v=5,col="orange")
```



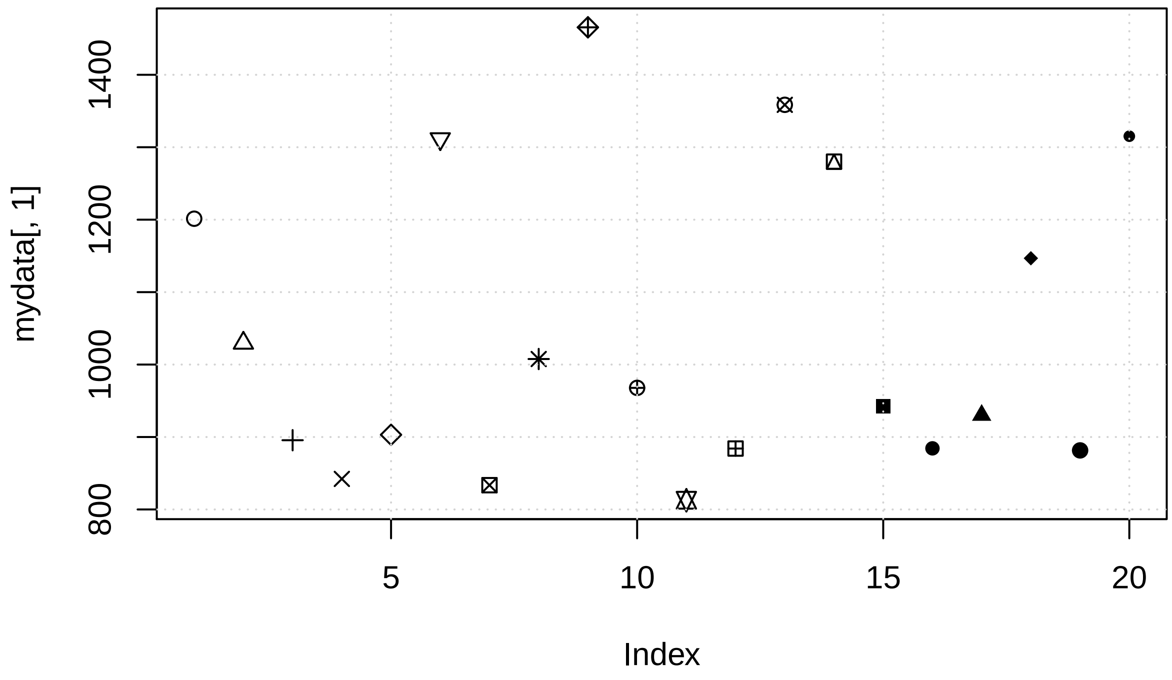
A dotted line be drawn by specifying `lty=2` (line type) and the width can be changed using the `lwd`(line width) argument.

```
plot(mydata[,1],pch=1:nrow(mydata))
abline(h=1000, col="steelblue", lty=2, lwd=5)
abline(v=5,col="orange",lty=3,lwd=10)
```



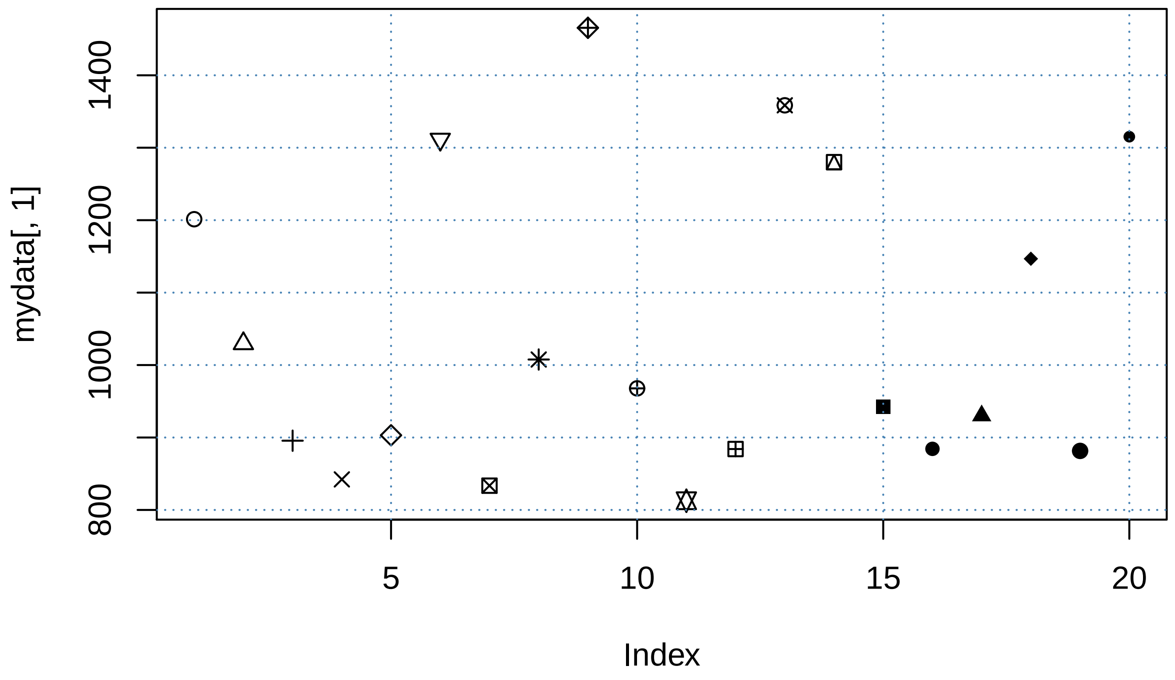
The `grid` function can be used to draw horizontal and vertical lines at regular intervals in the background.

```
plot(mydata[,1],pch=1:nrow(mydata))
grid()
```



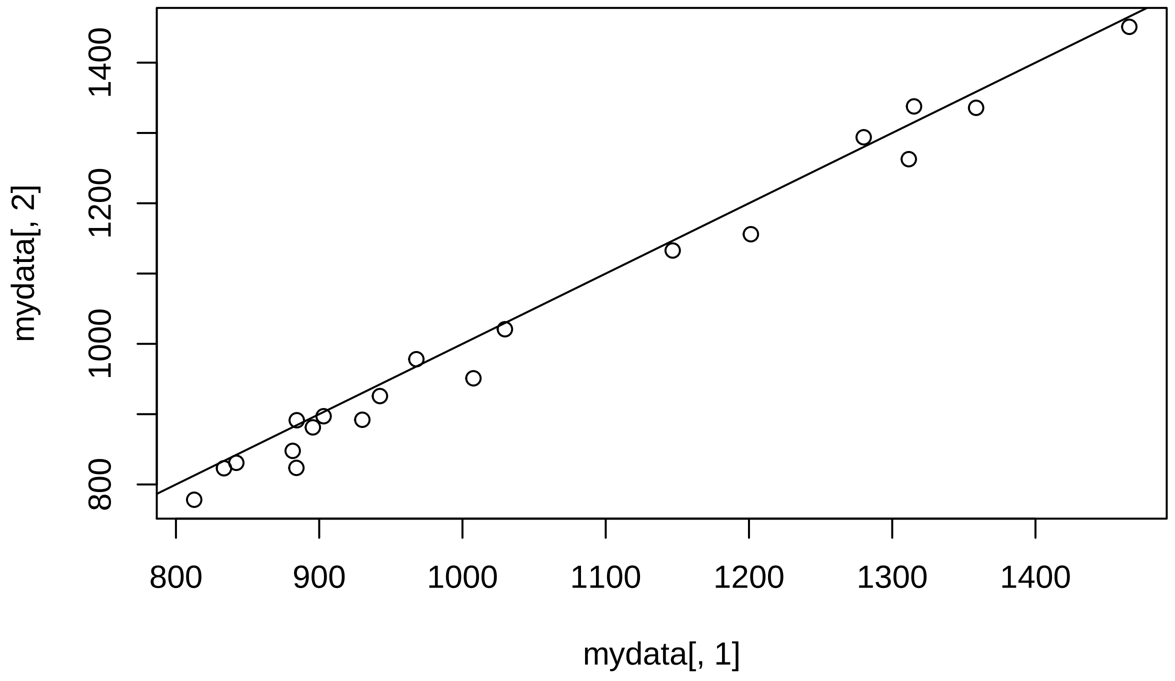
The colour of the grid lines can be changed in the usual way.

```
plot(mydata[,1],pch=1:nrow(mydata))
grid(col="steelblue")
```



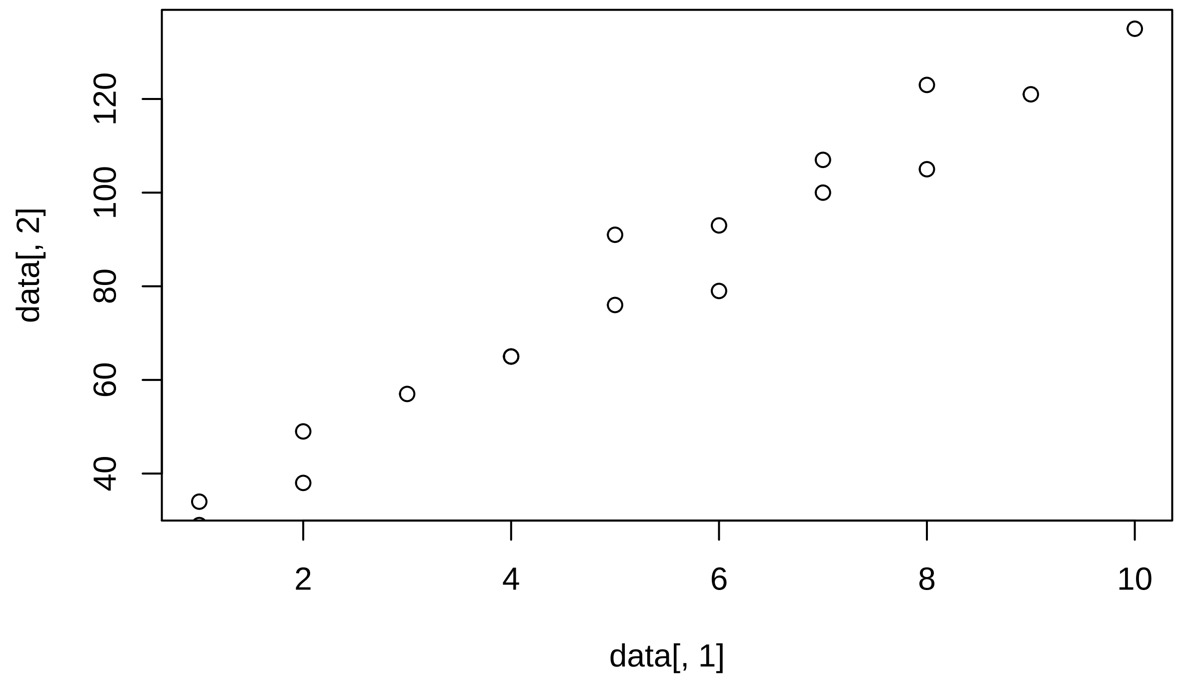
Another use for the `abline` function is to specify an intercept and coefficient for the line (the a and b in the `abline` name). For instance, if we wanted the line corresponding to equality of two vectors, $y = x$, we can say.

```
mydata <- read.csv("PrimerExamples/Paired two-sample t-test.csv")
plot(mydata[,1],mydata[,2])
abline(a=0,b=1)
```



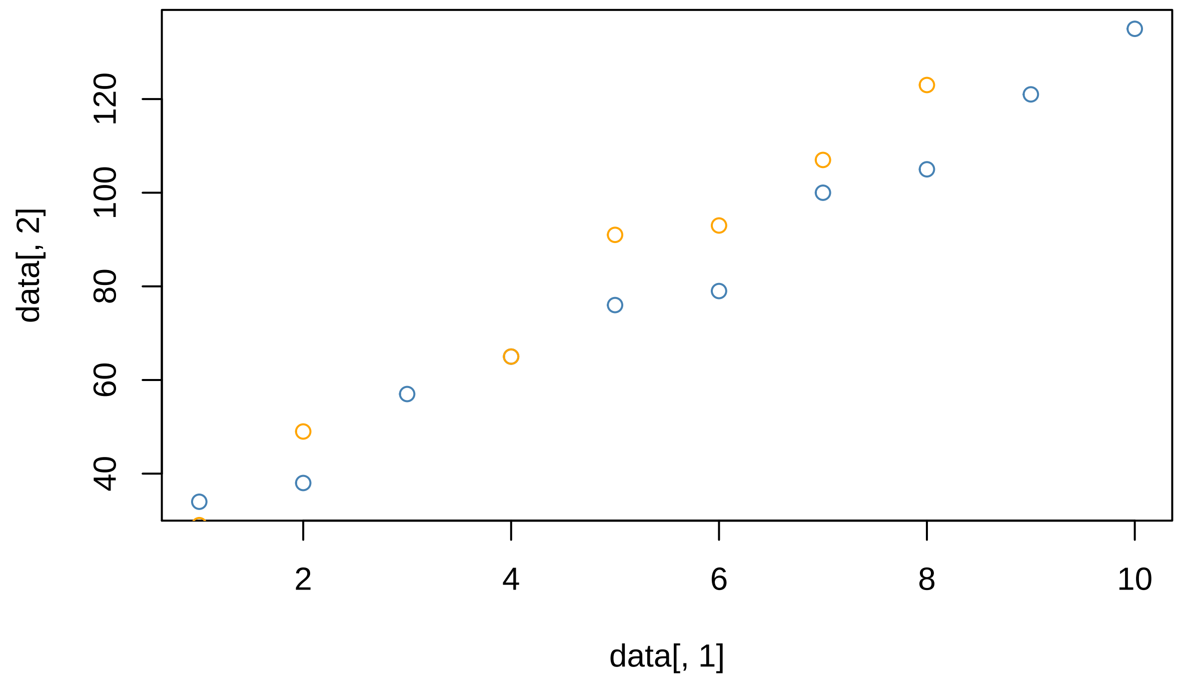
The `points` function can be used to add points to an existing plot. You need to specify vectors of x and y positions at which the points will be plotted.

```
data <- read.delim("PrimerExamples/Linear regression.txt")
plot(data[,1],data[,2])
points(data[,1],data[,3])
```



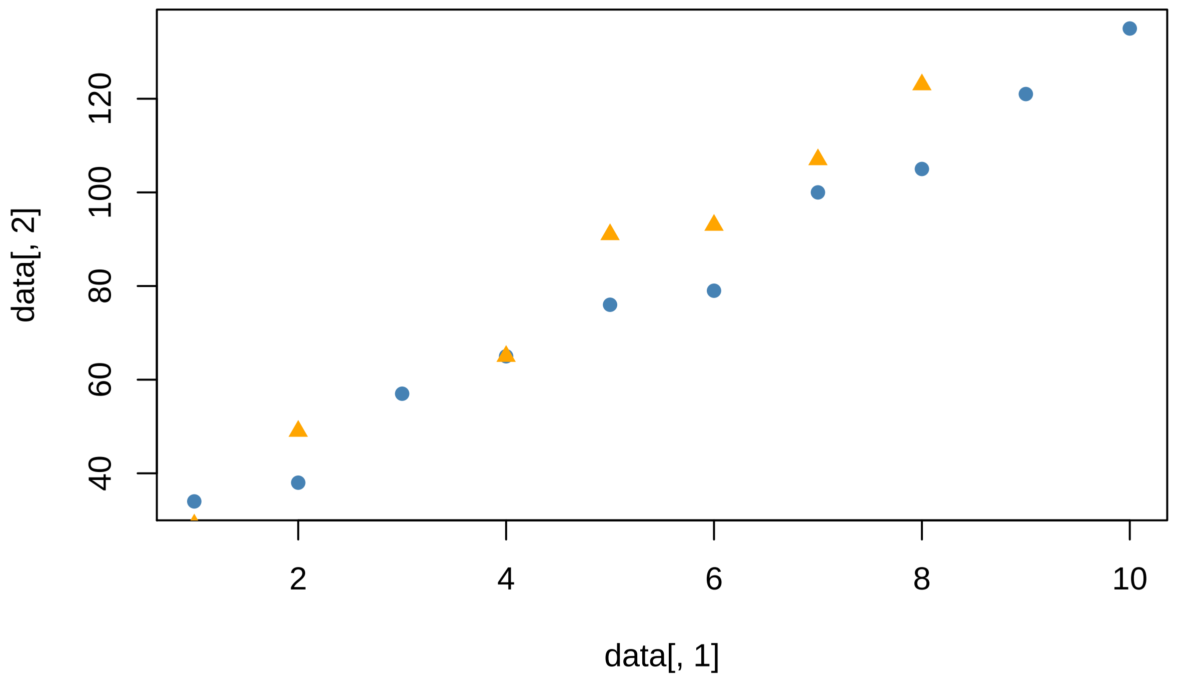
We can modify the colours in which points are plotted.

```
plot(data[,1],data[,2],col="steelblue")
points(data[,1],data[,3],col="orange")
```



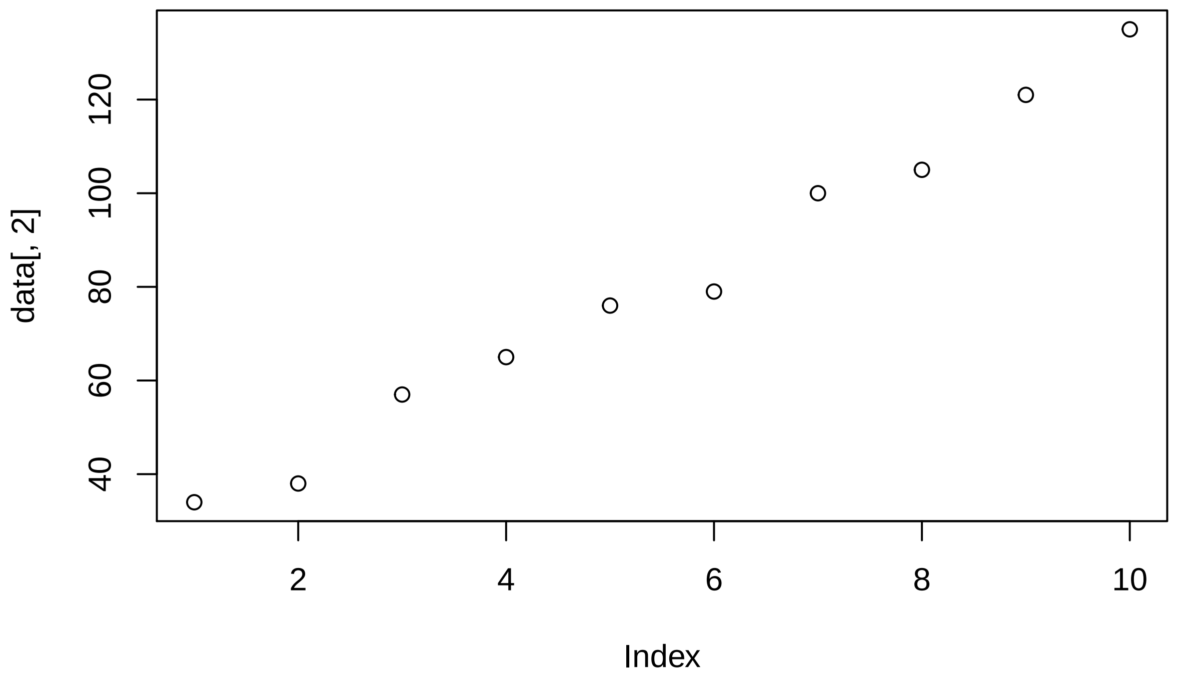
Or change the plotting characters

```
plot(data[,1],data[,2],col="steelblue",pch=16)
points(data[,1],data[,3],col="orange",pch=17)
```



The `arrows` function can be used to connect pairs of points on an existing plot. We have to supply *start* ane *end* x and y coordinates. In the example below, we draw and arrow from *(2,1.5)* to *(10,4)*.

```
plot(data[,2])
arrows(2, 1.5, 10, 4)
```

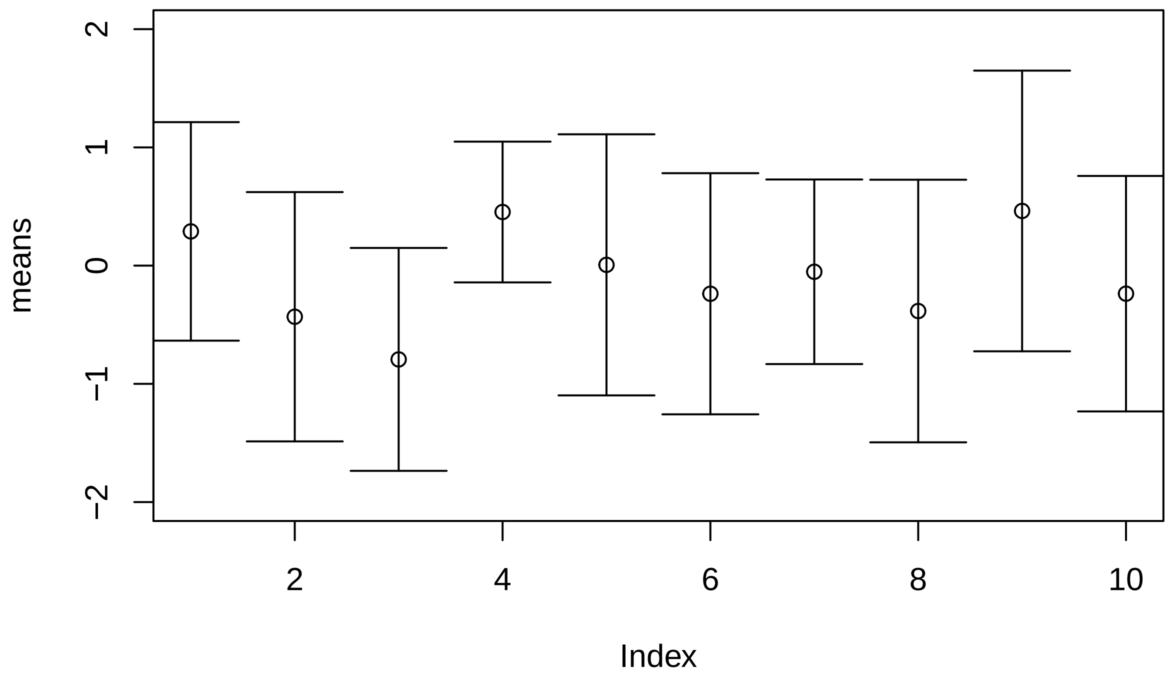


Customisations of the arrow are possible and multiple arrows can be drawn by supplying the coordinates as vectors. See the help page for more information.

```
?arrows
```

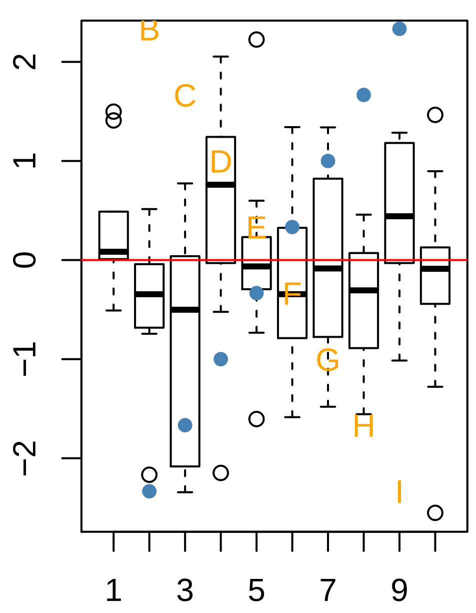
This function may not seem very useful initially, but it can be used as a means of drawing *error bars* on a plot.

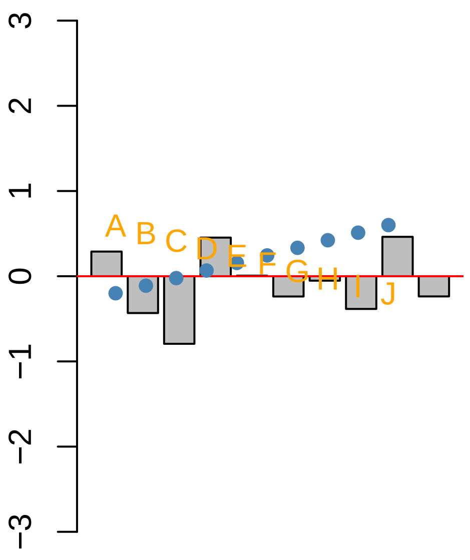
```
randMat <- matrix(rnorm(100),ncol=10)

means <- colMeans(randMat)
plot(means,ylim=c(-2,2))

err <- apply(randMat, 1, sd)

arrows(1:10, means-err,1:10,means+err,code=3,angle=90)
```



So far in this section, we have shown how to modify scatter plots using `lines`, `points` etc. However, the same modifications can be applied to other types of plot too.

```
par(mfrow = c(1,2))
# Boxplot
boxplot(randMat)
abline(h = 0,col="red")
points(1:10, y = seq(-3,3,length.out = 10),pch=16,col="steelblue")
text(1:10, y= seq(3,-3,length.out = 10),labels = LETTERS[1:10],col="orange")

# Barplot
barplot(colMeans(randMat),ylim=c(-3,3))
abline(h = 0,col="red")
points(1:10, seq(-0.2, 0.6, length.out=10),col="steelblue",pch=16)
text(1:10, y= seq(0.6,-0.2,length.out = 10),labels = LETTERS[1:10],col="orange")
```



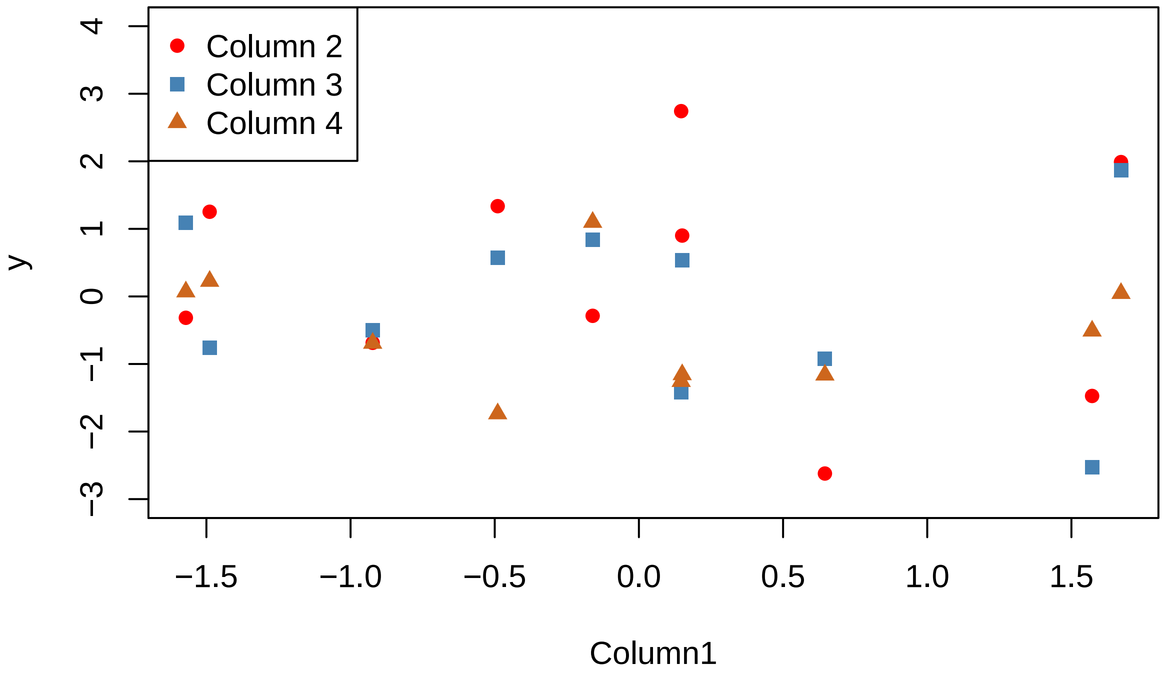


Adding a legend

The `legend` function can be used to add a legend to an existing plot. In the following example, we generate a random matrix and plot the second, third and fourth columns against the first. We use red circles, blue squares and orange triangles respectively to represent these variables and we will use the legend to indicate this mapping.

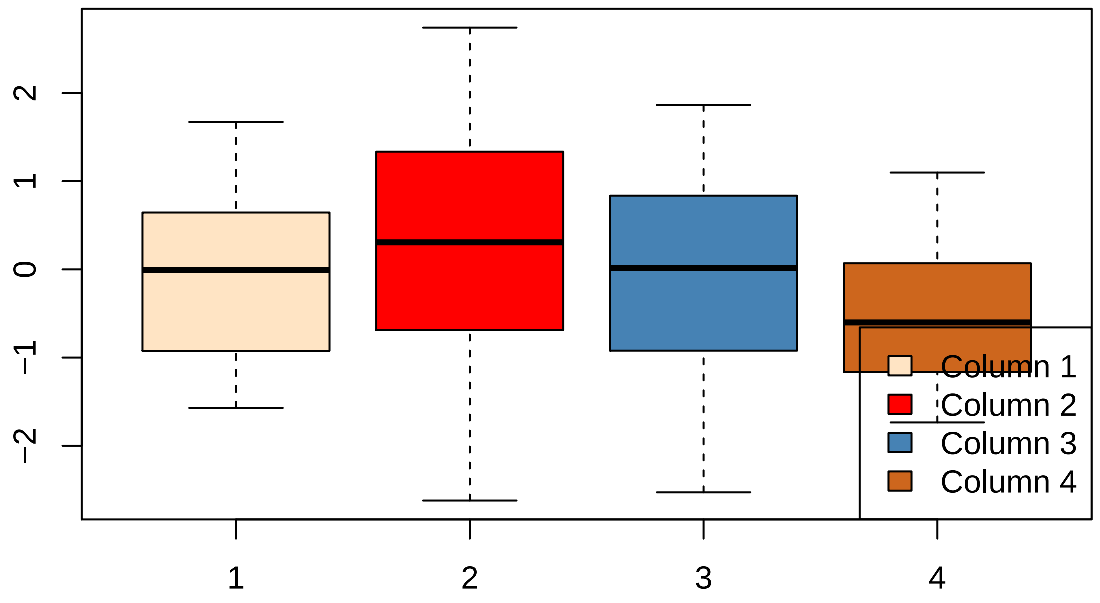
The names to be used in the legend are defined by the `legend` argument, while `col` and `pch` can be used to define the relevant colours and plotting characters in the same way as they are in the plot itself. For fine control over the position of the legend we can specify the x and y coordinates of where the legend will appear. However this can be tedious if you change the size of the plot, so the strings `topleft`, `topright`, `bottomleft` and `bottomright` are conveniant shortcuts for the most common locations.

```
randMat <- matrix(rnorm(100),ncol=10)
plot(randMat[,1],randMat[,2],ylim=c(-3,4),pch=16,col="red",xlab="Column1",ylab="y")
points(randMat[,1],randMat[,3], pch=15,col="steelblue")
points(randMat[,1],randMat[,4], pch=17,col="chocolate3")

legend(x = "topleft", legend=c("Column 2", "Column 3","Column 4"), col = c("red", "steelblue", "chocola
```



A legend can be added to all types of plots; as shown here for a boxplot. Unlike the previous example, here we're not interested in the shape off the plotting character and only wanted to indicte the colour in the legend. To achieve this we need to use `fill` argument instead of the combination of `pch` & `col` we used before.

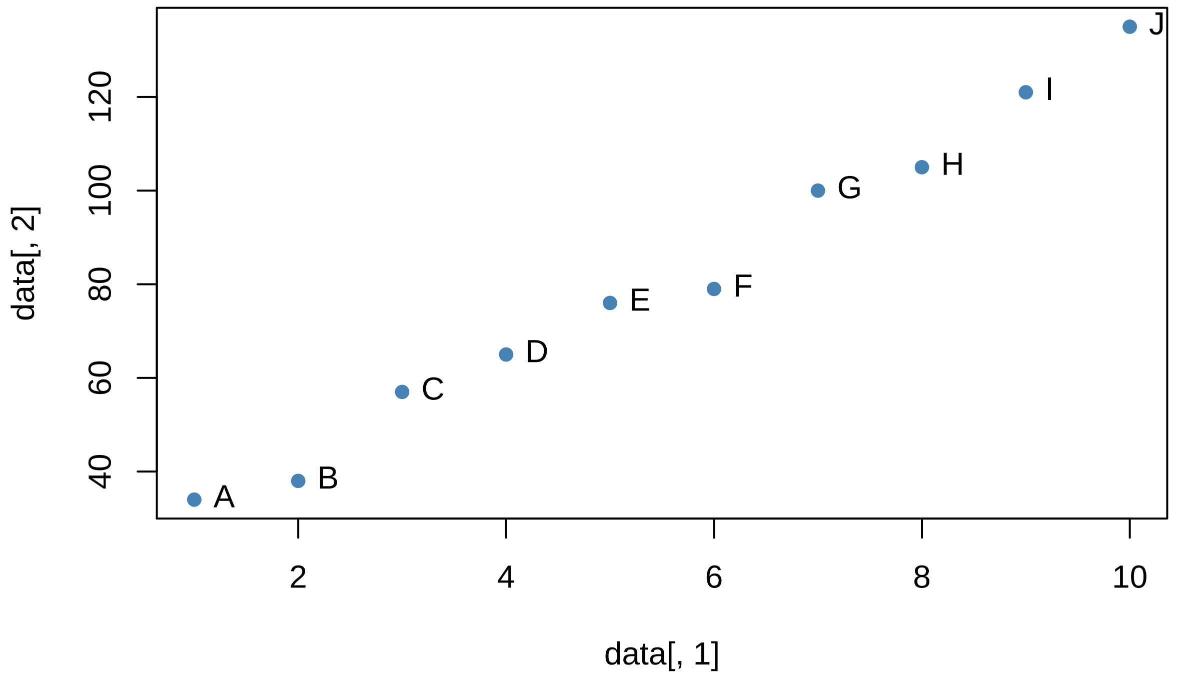
```
boxplot(randMat[,1:4],col=c("bisque1", "red", "steelblue","chocolate3"))
legend("bottomright", legend=c("Column 1", "Column 2", "Column 3","Column 4"), fill = c("bisque1","red"
```



Adding text to plots

A common task is to add labels to points in a plot. This can be achieved using the function `text()`, which adds text to an existing plot. In the example below we first plot 10 points. These same values are used in the `text` command to position the labels. The argument `pos = 4` indicates we want to text to be placed to the right of the specified coordinates; if we didn't specify this the label would cover the points. The in this example the labels are the first 10 letters of the alphabet, but the could equally be longer words such as the names of samples.

```r
plot(data[,1], data[,2], col="steelblue", pch=16)
text(x = data[,1], y = data[,2], pos = 4, labels=LETTERS[1:10])
```



Sometimes, we want to add annotations to the plot outside of the plotting window. This can be achieved by using the `mtext` function. We use the paste function to construct our labels and have to specify the x coordinates at which to put the text. The `line` argument specifies how far from the border of the plot to plot to write the text.

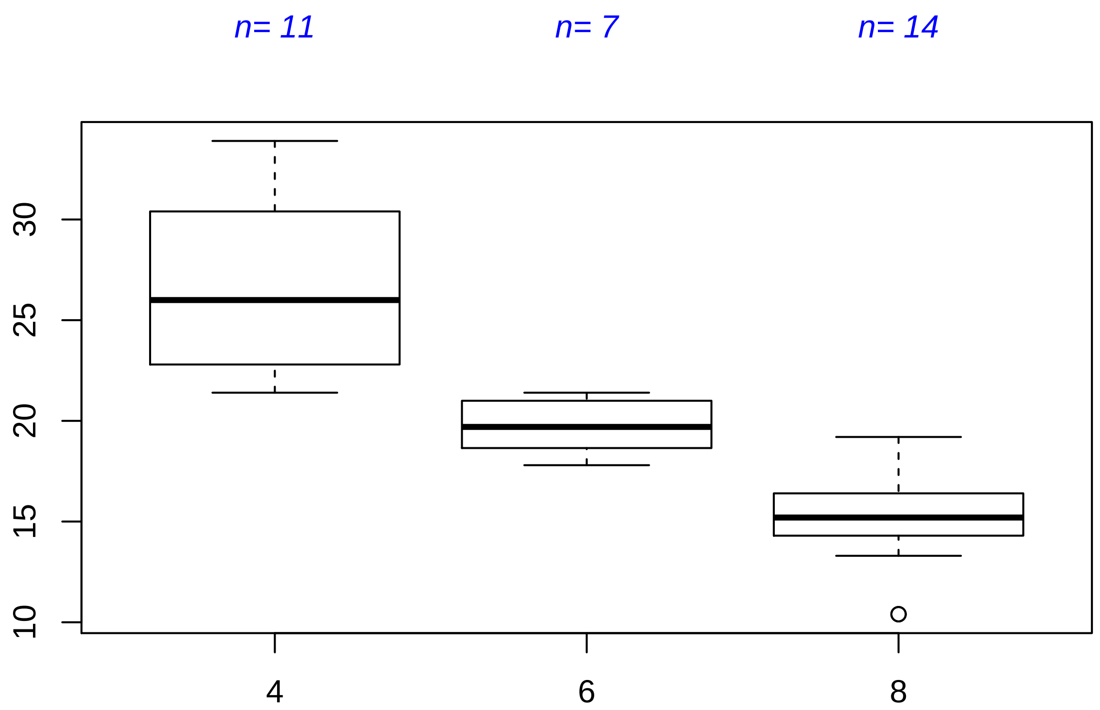
```r
data(mtcars)
boxplot(mtcars$mpg~mtcars$cyl)
table(mtcars$cyl)
```

```
##
##  4  6  8
## 11  7 14
```

```r
labs <- paste("n=", table(mtcars$cyl))
labs
```

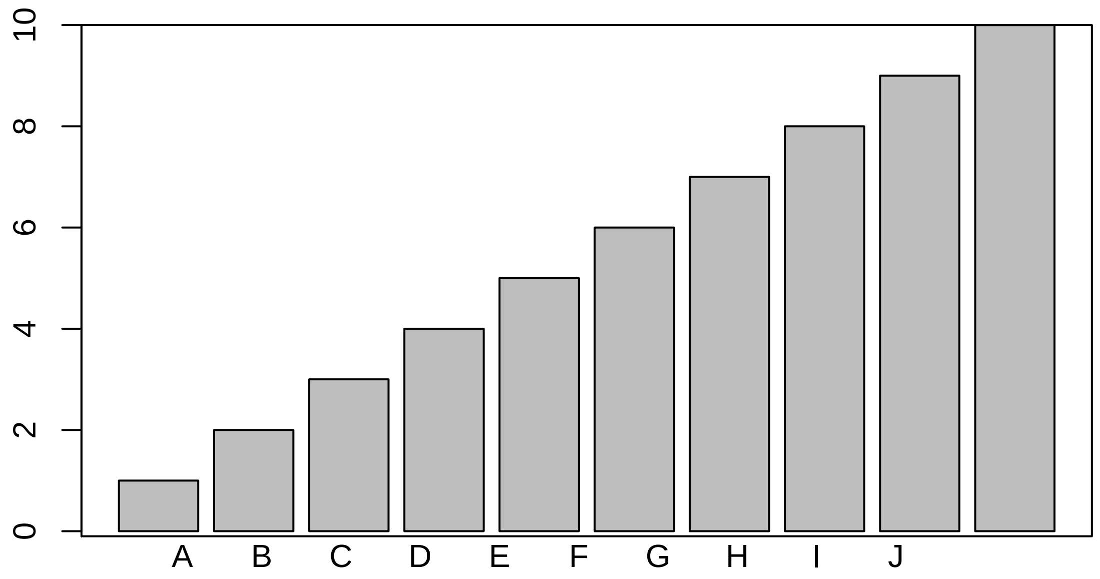
```
## [1] "n= 11" "n= 7"  "n= 14"
```

```r
mtext(side=3, at=1:3,text=labs,line=2,col="blue",font=3)
```



In the above example, we set the x coordinates to be 1,2,3 because we had three groups in the boxplot. Unfortunately, the `barplot` function does not place the centers of the bars in such a uniform manner.

```
barplot(1:10)
box()
mtext(side=1, at=1:10,text=LETTERS[1:10])
```

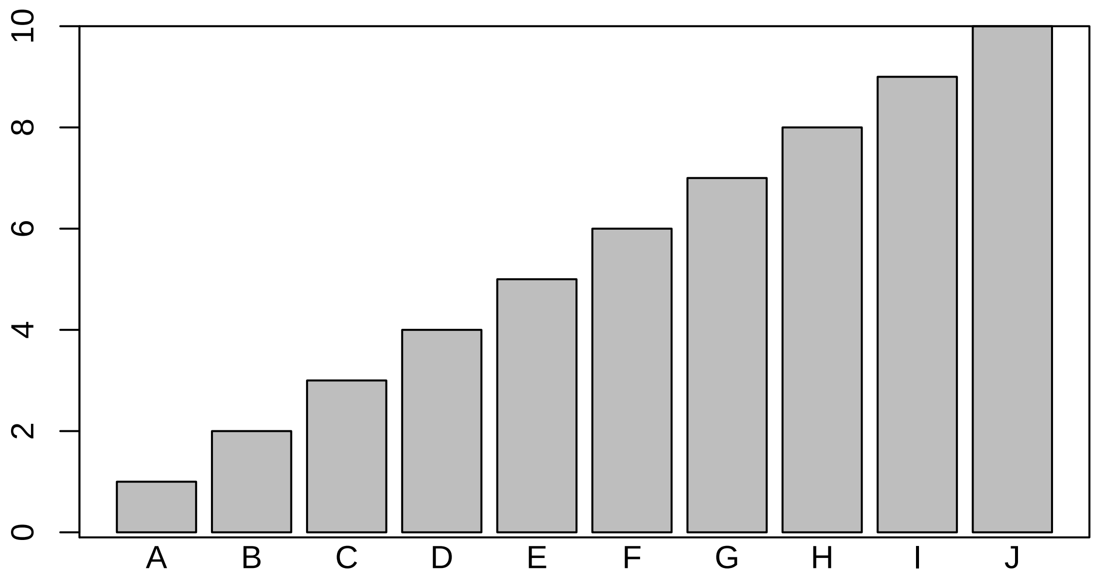


To get around this, we can assign the result of `barplot` to a variable. This variable then contains the midpoints of each bar, which we can then use as input for the `mtext` function.

```
bp <- barplot(1:10)
bp
```

```
##       [,1]
##  [1,]  0.7
##  [2,]  1.9
##  [3,]  3.1
##  [4,]  4.3
##  [5,]  5.5
##  [6,]  6.7
##  [7,]  7.9
##  [8,]  9.1
##  [9,] 10.3
## [10,] 11.5
```

```
box()
mtext(side=1, at=bp,text=LETTERS[1:10])
```



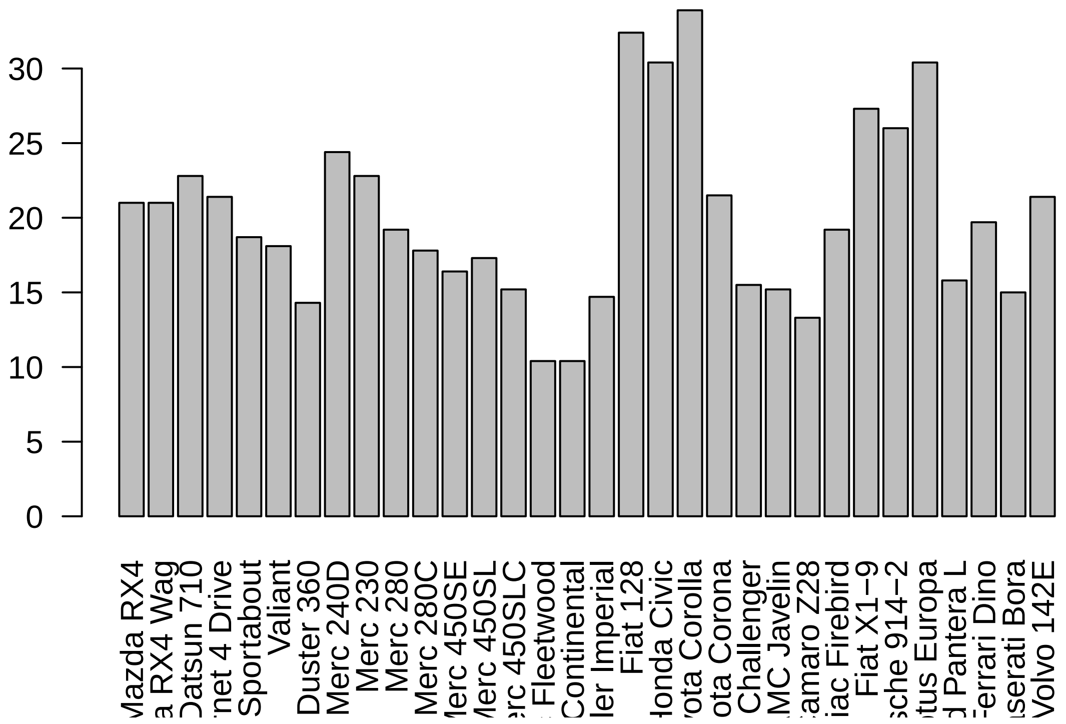
The par function

The par function can be used to alter various graphical parameters. These parameters will be applied to the next plot to be created. We will now go through some of the most-popular uses of the par function. For a comprehensive list of parameters that can be changed, see the help page for `par`; `?par`.

Adding extra space around a plot

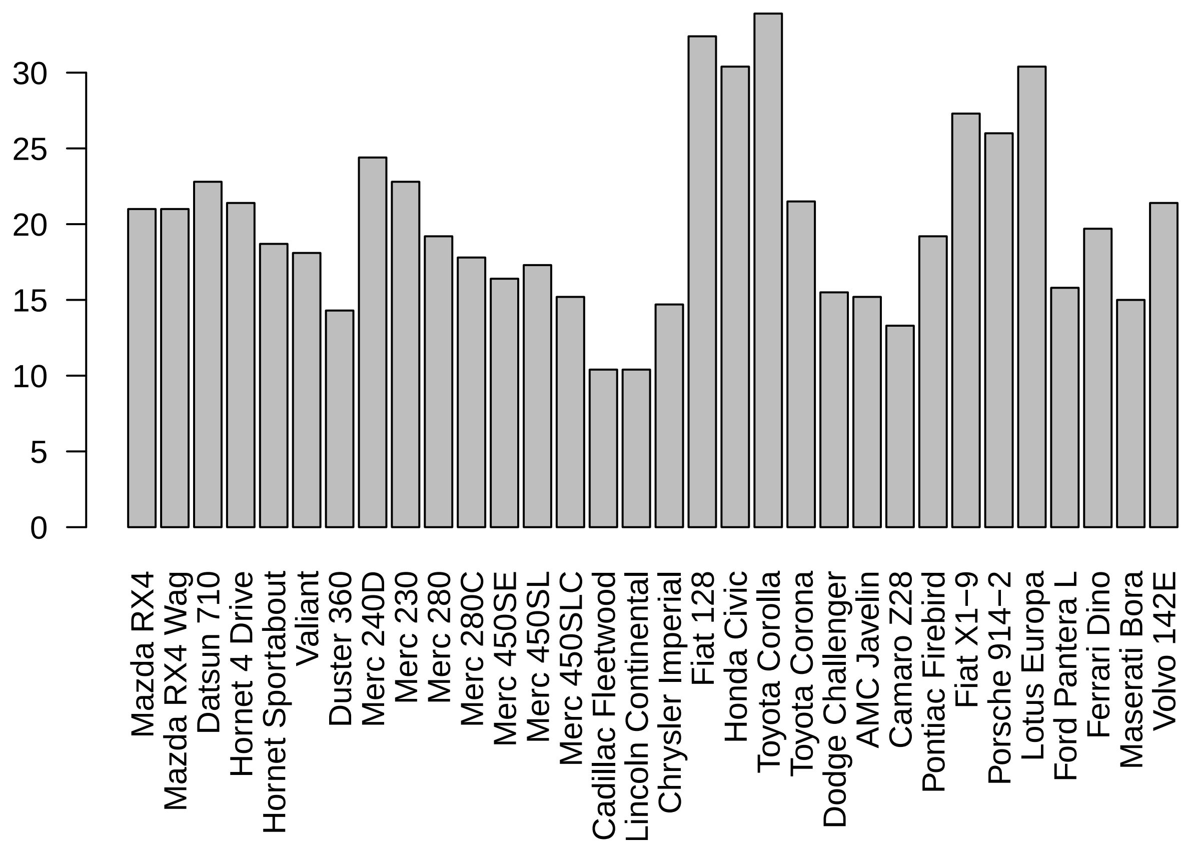
Taking the `mtcars` dataset as an example, we can sometimes create a plot where they is not adequate space to fit all the labels. Here we make a barplot of the `mpg` variable and modify the axes. However, we see there is not enough room to fit the labels

```
data(mtcars)
barplot(mtcars$mpg,names.arg = rownames(mtcars),las=2)
```



We can use the `par` function before the plot is created to give us more room. The variable we need to change is `mar` which should be a vector of length 4; referring to the amount of space 1) below 2) left 3) above 4) to the right. The default setting is `c(1,1,1,1)` which gives one line of space at each edge of the plot. In this example, we allow more lines below the plot.

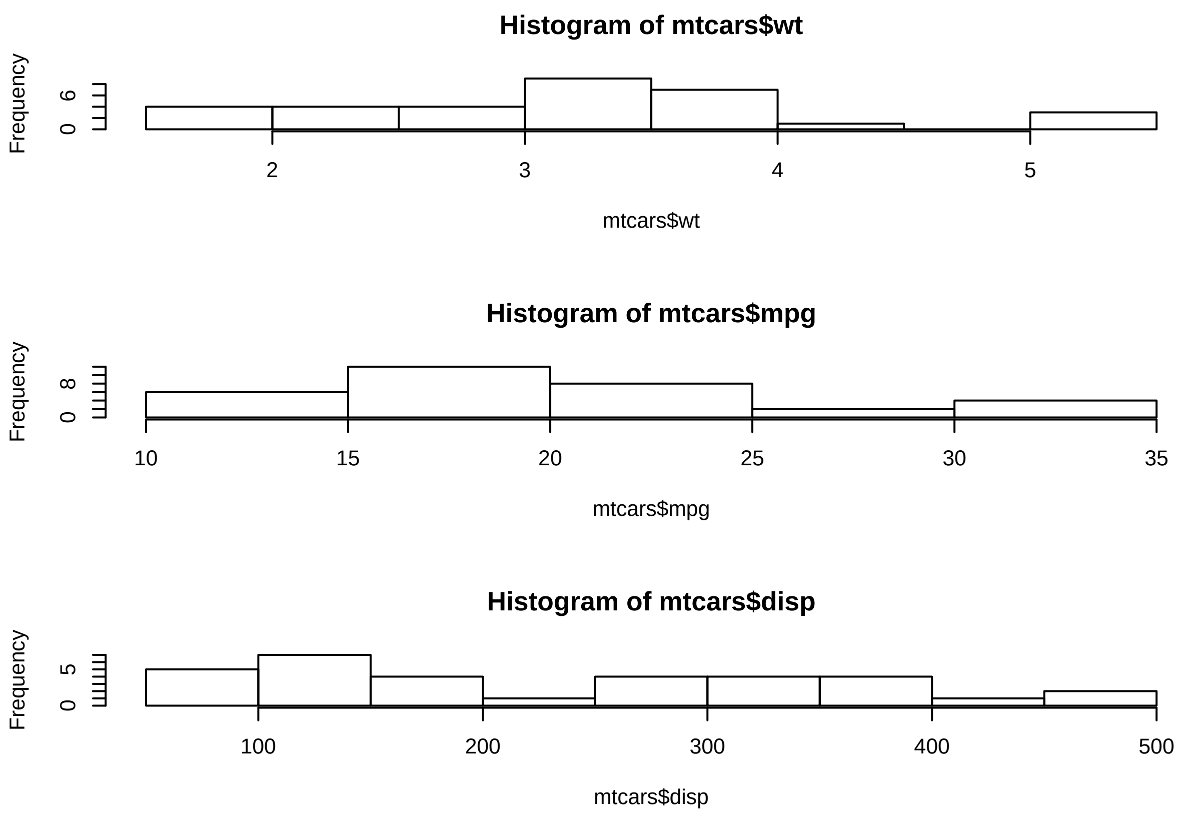
```r
data(mtcars)
par(mar=c(8,2,1,1))
barplot(mtcars$mpg,names.arg = rownames(mtcars),las=2)
```



Combining multiple plots

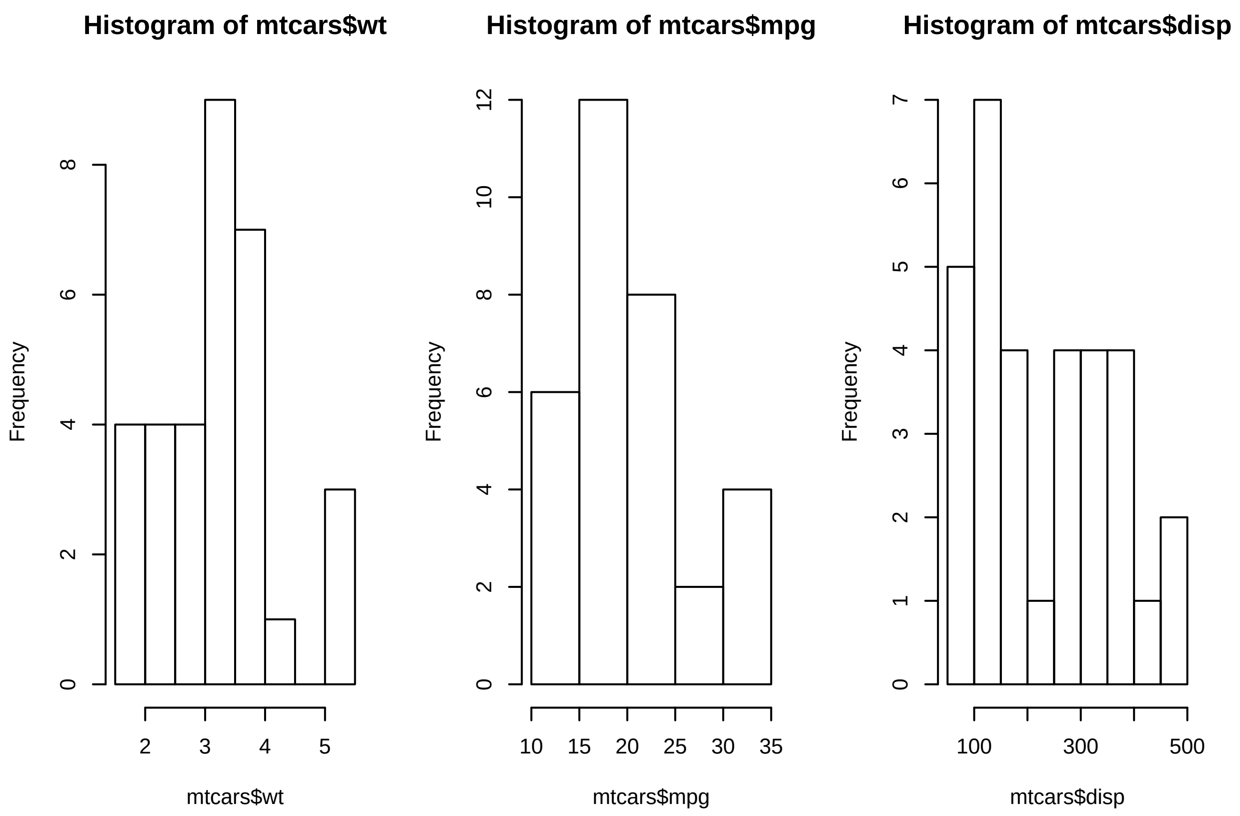
The `mfrow` variable can be set by the `par` function, which will change the layout of future plots. It requires a vector of length 2 corresponding to how many *rows* and *columns* the plotting device is divided into. The following arranges the plotting window into 3 rows and 1 column

```
data(mtcars)
par(mfrow=c(3,1))
hist(mtcars$wt)
hist(mtcars$mpg)
hist(mtcars$disp)
```



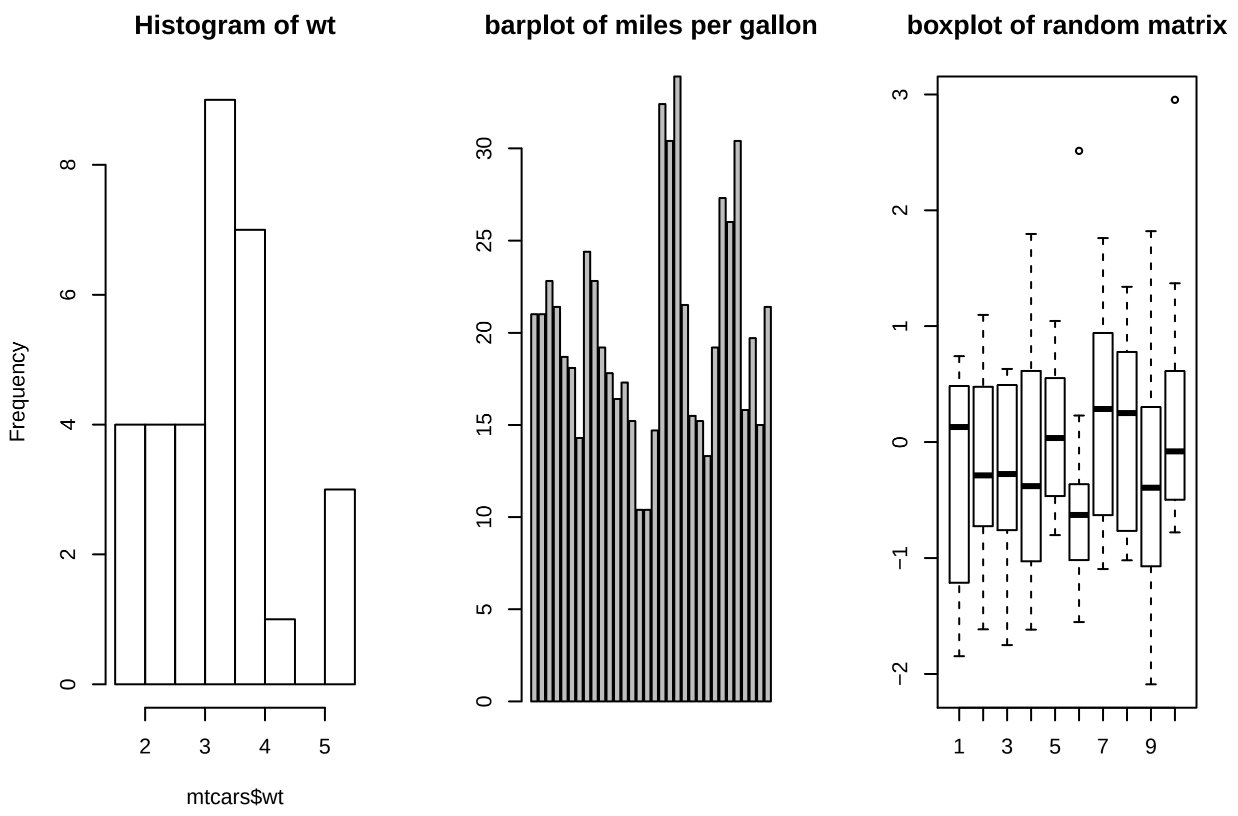
Similarly, we could have 1 row and 3 columns;

```r
data(mtcars)
par(mfrow=c(1,3))
hist(mtcars$wt)
hist(mtcars$mpg)
hist(mtcars$disp)
```



We are not restricted to having the same type of plot, or even the same dataset plotted in each cell.

```r
data(mtcars)
randMat <- matrix(rnorm(100),ncol=10)
par(mfrow=c(1,3))
hist(mtcars$wt,main="Histogram of wt")
barplot(mtcars$mpg, main="barplot of miles per gallon")
boxplot(randMat, main="boxplot of random matrix")
```



Statistical Analysis

Appendix

Vectors

Vectors are the fundamental data type in R and are composed of an ordered group of single items of data. There are several ways to construct a vector, but perhaps the simplest and most commonly used is `c()` function. This may seem like an unintuative name (you'll get used to some of R's idosyncracies if you use it long enough), but it stands for *concatenate* as what the function is actually doing is sticking together a number of smaller objects.

```
numericVector <- c(1, 2, 3, 4)
```

When you create a variable with a single value, you are still creating a vector - it just has a length of one. This is why we can describe vectors as the fundamental data type in R, there is nothing smaller to break them down into.

```
shortVariable <- 1
shortVariable
```

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

```
shortVariable[1]
```

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

Vectors don't have to contain numbers, they work equally well for storing character data.

```
characterVector <- c("one", "two", "three", "four")
```

One important thing to note is the all the values in a vector must be of the same type. For instance you cannot have numeric and character data in the same vector. However R won't neccessarily report and error if you try to do this, as the example below shows:

```
mixedVector <- c(1, 2, "three", "four")
mixedVector
```

```
## [1] "1"     "2"     "three" "four"
```

You can see in the output that all four values have quotation marks around them. This indicates that R has automatically converted the numeric value to character strings. If you wanted to keep the distinct data types you may wish to use a data frame, which we'll discuss later.

Matrices

Sometimes it's useful to store values in two dimensions (rows and columns) rather than in a linear fashion as we can with vectors. For example if you have an experiment with several samples and multiple time points it may make sense to store data in a format where the rows are samples and columns represent times.

In order to achieve this we can use a matrix object. The code below generates two small matrices, each containing the number 1 to 16.

```
matrix1 <- matrix(1:16, nrow = 4)
matrix1
```

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

```
matrix2 <- matrix(1:16, nrow = 2)
matrix2
```

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

The `nrow` argument allows use to specify the dimensions of the matrix we create. Similarly we can specify the argument `byrow` to change the order in which the values are placed in the matrix. The default value of this agument is `FALSE`, specifying that the matrix is filled going down the columns. In the example below we change this behaviour and fill and the values are placed going across columns instead.

```
matrix(1:16, nrow = 2, byrow = TRUE)
```

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

As with vectors, matrices can only contain a single type of data; you can't mix numbers and characters in the same matrix.

Data Frames

If we want to store multiple types of data in a single object with the row and column structure you might expect from a spreadsheet or printed table, we need to use a new object type: the *data frame*. In the example below we'll create a data frame using two of the vectors created in the previous section.

```
charNumFrame <- data.frame(numericVector, characterVector)
charNumFrame
```

```
##   numericVector characterVector
## 1             1             one
## 2             2             two
## 3             3           three
## 4             4            four
```

Lists

Although it's easy to think of vectors as 'lists' of values, within R a list is a distinct type of object. Lists are used when you want to group together disparate types of information. For example, one could store several vectors of numbers, but they need not be the same length, something that a matrix or data frame could not cope with. Alternatively, a list could contain four elements that are a vector, a matrix, a data frame and another list respectively.

```
list1 <- list(vec1 = 1:2, vec2 = 3:5, vec3 = 6:9, vec4 = 10:13)
list2 <- list(vec = 1:10, mat = matrix(1:10, nrow = 2), dat = data.frame(a = 1:5, b = 6:10), lis = list
```

Subsetting

Sometimes you only want to use a subset of the data stored in an object; perhaps you only want the first five items from a long vector, or you're only interested in the second column from a data frame. To extract a portion of an object we need to use R subsetting functionality, which is denoted by square brackets `[]`.

```
characterVector[2:3]
```

```
## [1] "two"   "three"
```

```
matrix1[3:4, ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3    7   11   15
## [2,]    4    8   12   16
```

In the example above the first line extracts the 2nd and 3rd entries from a vector, while the second line selects rows 3 and 4 from a matrix. If we wanted to obtain the same rows, but only for the first column we'd need to specify that too:

```
matrix1[3:4, 1]
```

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

If you're really paying attention to the example above you'll notice that when we only select one column the format the data is printed to screen in changes. The entries are now next to each other rather than on top of one another. This is because when you only select one column R automatically gives you back a vector rather than a matrix (it assumes you don't need the second dimension). This behaviour can sometimes be useful and sometimes frustrating, but if you want to ensure it doesn't happen you can use the `drop = FALSE` argument seen below.

```
matrix1[3:4, 1, drop=FALSE]
```

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

Extracting subsets from a list is slightly more subtle than for other object types. Where as before we used a single set of square brackets `[ ]`, lists make use of both this and a double set of brackets `[[ ]]`.

When using a single set of brackets on a list, R will return another (probably shorter) list. In the code below we ask for a list with only the first and third elements (the vector and data frame entries) from the example made earlier.

```
list2[c(1,3)]
```

```
## $vec
##  [1]  1  2  3  4  5  6  7  8  9 10
## 
## $dat
##   a  b
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

This makes sense if you're interested in getting a few entries but can be a little unintuative if you're only interested in a single element. Asking for a single entry in the manner shown above will still return a list. If you wish to reference an element directly you use the double braket notation. This difference is demonstrated in the example below, where we use the function `class()` to ask R what type of object we've got.

```
class( list1[1] )
```

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

```r
class( list1[[1]] )
```

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

Functions

Useful functions to know

R version details

The version of R and the packages used in this document are given below.

```r
sessionInfo()
```

```
## R version 3.1.0 (2014-04-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=en_GB.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_GB.UTF-8        LC_COLLATE=en_GB.UTF-8
##  [5] LC_MONETARY=en_GB.UTF-8    LC_MESSAGES=en_GB.UTF-8
##  [7] LC_PAPER=en_GB.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_GB.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] crukCIMisc_0.2     RColorBrewer_1.0-5 foreign_0.8-61
## [4] gdata_2.13.3
##
## loaded via a namespace (and not attached):
##  [1] digest_0.6.4     drc_2.3-96       evaluate_0.5.5   formatR_0.10
##  [5] gtools_3.4.0     htmltools_0.2.4  knitr_1.6        plyr_1.8.1
##  [9] Rcpp_0.11.1      reshape2_1.4     rmarkdown_0.2.50 splines_3.1.0
## [13] stringr_0.6.2    survival_2.37-7  tools_3.1.0      yaml_2.1.12
```