Module Two

Lesson 1: Data Ojects: List and Data Frame

Data Objects

In R, the data are stored as objects.

All data objects generally have three properties.

- A variable name
- A type (or class): scalar, vector, matrix, etc.
- Values correponding to their type

Array

We have already seen two-dimensional arrays which are the same thing as matrices.

One-dimensional arrays look like the vectors.

Generally we can have d-dimensional arrarys. The following examples shows a three-dimensional 2 by 2 by 2 array.

```
array(1:8,c(2,2,2)) # A 2 by 2 by 2 array, which consists of two matrices of size 2 by 2.
```

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

List

A **list** is a generic vector containing other objects. Lists are useful for organizing information. For example, the following array contains a numeric vector, a character strings vector, a matrix with Boolean elements, and a numer.

```
mynum = c(2,4) # A vector of 2 numbers
mystr = c("I", "love", "stat") # A vector of 3 character strings
myboo = matrix(c(TRUE, FALSE, FALSE, TRUE), ncol=2) # A 2x2 matrix with boolean elements
mylist = list(mynum, mystr, myboo, 100) #joint them into a list
mylist # display the list
```

```
## [[1]]
## [1] 2 4
##
## [[2]]
## [1] "I"    "love" "stat"
##
## [[3]]
##       [,1]  [,2]
## [1,]  TRUE FALSE
## [2,] FALSE  TRUE
##
## [[4]]
## [1] 100
```

The above variable *mylist* is a list containing copies of two vectors *mynum*, *mystr*, the matrix *myboo*, and a numeric value *100*. The elements of *mylist* are copies of those vectors and the matrix. That is, if you modify the elements of *mylist*, the original vectors *mynum*, *mystr*, and the matrix *myboo* will not change.

List Slicing

We retrieve a list slice with the single square bracket "[]" operator. The following is a slice containing the second member of mylist.

```
mylist[2] # A new one-element list containing the second member of mylist
```

```
## [[1]]
## [1] "I"    "love" "stat"
```

We can retrieve a slice with multiple members, with an index vector. Here a slice containing the second and fourth members of mylist.

```
mylist[c(2, 4)] # A new two-elements list containing the second and fourth members of mylist
```

```
## [[1]]
## [1] "I"    "love" "stat"
##
## [[2]]
## [1] 100
```

Member Reference

In order to reference a list member directly, we use the double square bracket "[[]]" operator. The following object mylist[[2]] is the second member of *mylist*.

```
mylist[[2]] # A character strings vector, copy of mystr which is the second element in mylist
```

```
## [1] "I"    "love" "stat"
```

We can modify the content of the memeber directly.

```
mylist[[2]][1] = "You"
mylist[[2]] # The second element in the list is changed. The change is done on the copy of mystr
```

```
## [1] "You"  "love" "stat"
```

```
mystr # The original character strings vector remains the same
```

```
## [1] "I"    "love" "stat"
```

Data Frame

A **data frame** is frequently used for storing data tables. It is a list of vectors with **equal** length. You can think of a data frame object as a being somewhat like a spreadsheet. Each column of the data frame is a vector. Within each vector, all data elements must be of the same mode (numeric, characters, etc). However, different vectors can be of different modes.

For example, the following variable `mydf` is a data frame containing three vectors `mynum`, `mystr`, and `myboo`.

```
mynum = c(2,4,6) # A vector of 2 numbers
mystr = c("I", "love", "stat") # A vector of 3 character strings
myboo = c(TRUE, FALSE, TRUE)
mydf = data.frame(mynum, mystr, myboo)
mydf
```

```
##   mynum mystr myboo
## 1     2     I  TRUE
## 2     4  love FALSE
## 3     6  stat  TRUE
```

You can think each column of the data frame as one variable. We refer to a specific variable in the data frame, using the “$” operator.

```
mydf$mystr       # display the 'mystr' vector in the 'mydf' data frame
```

```
## [1] I    love stat
## Levels: I love stat
```

```
mydf[,'mystr']   # Same as taking the column 'mystr' from the 'mydf' matrix
```

```
## [1] I    love stat
## Levels: I love stat
```

```
mean(mydf$mynum) # calcualte the average of the 'mynum' vector in the 'mydf' data frame
```

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

You can simply treat a data frame as a matrix. The only difference is that you can refer to the columns in the data frame by the “$” operator.

Lesson 2: Working with Data Objects

Creating and Changing data matrices

Commonly we may want to combine vectors into a matrix. Functions rbind, for rows, and cbind, for columns, easily perform these tasks.

```
gene1 <- c(1.00,1.50,1.25)
gene2 <- c(1.35,1.55,1.00)
cbind(gene1,gene2) # combine the two vectors, each as a column. Result in a 3 by 2 matrix
```

```
##      gene1 gene2
## [1,]  1.00  1.35
## [2,]  1.50  1.55
## [3,]  1.25  1.00
```

```
rbind(gene1,gene2) # combine the two vectors, each as a row. Result in a 2 by 3 matrix
```

```
##       [,1] [,2] [,3]
## gene1 1.00 1.50 1.25
## gene2 1.35 1.55 1.00
```

Note these functions work for matrices as well.

```
gene1 <- c(1.00,1.50,1.25)
gene2 <- c(1.35,1.55,1.00)
gene12<-rbind(gene1,gene2) # combine the two vectors, each as a row. Result in a 2 by 3 matrix
gene12 # print the 2 by 3 matrix
```

```
##       [,1] [,2] [,3]
## gene1 1.00 1.50 1.25
## gene2 1.35 1.55 1.00
```

```
gene3 <- c(-1.10,-1.50,-1.25)
gene4 <- c(-1.20,-1.30,-1.00)
gendat<-rbind(gene12,gene3,gene4) # add two more rows to the 2 by 3 matrix. Result in a 4 by 3 matrix
colnames(gendat)<-c("Eric","Peter","Anna") # Name the colums
gendat # print the 4 by 3 matrix
```

```
##        Eric Peter  Anna
## gene1  1.00  1.50  1.25
## gene2  1.35  1.55  1.00
## gene3 -1.10 -1.50 -1.25
## gene4 -1.20 -1.30 -1.00
```

Check Types

Sometimes you may forget or not know what type of data you are dealing with, so R provides functionality for you to check this. There is a set of “is.what” functions, which provide identification of data object types and modes. For example:

```
#checking data object type
is.vector(gendat) # is gendat a vector?
```

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

```
is.matrix(gendat) # is gendat a matrix?
```

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

```
is.data.frame(gendat) # is gendat a data frame?
```

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

```
#checking data mode
is.numeric(gendat) # are the elements in gendat numbers?
```

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

```
is.character(gendat) # are the elements in gendat character strings?
```

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

The ‘str()’ function can give all information including types, modes on a data object.

```
str(gendat) # display info about gendat. From below, we can see that gendat has numbers in a two-dimens
```

```
##  num [1:4, 1:3] 1 1.35 -1.1 -1.2 1.5 1.55 -1.5 -1.3 1.25 1 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:4] "gene1" "gene2" "gene3" "gene4"
##   ..$ : chr [1:3] "Eric" "Peter" "Anna"
```

Change Types

There is a set of “as.what” functions, which change data object types and modes. For example, you may want to convert a vector into a matrix:

```
y<-as.matrix(c(1,2,3,4)) # The vector become a 4 by 1 matrix (a column vector)
y
```

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

If you want a matrix other than the column vector, then you need to use *matrix()* function, not *as.matrix()* function.

```
as.matrix(c(1,2,3,4),nrow=2) # The vector become a 4 by 1 matrix (a column vector). nrow=2 is ignored
```

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

```
matrix(c(1,2,3,4),nrow=2) # The vector become a 2 by 2 matrix
```

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

You may also change a numerical vector to a character mode vector.

```
z<-as.character(gene1) # The numbers are coded as character strings.
z
```

```
## [1] "1"    "1.5"  "1.25"
```

R is smart enough to try catching you if you try to do an illogical conversion, such as convert character data to numeric mode. It does do the conversion but the data is converted to NA values.

```
mystr
```

```
## [1] "I"    "love" "stat"
```

```
as.numeric(mystr) # Changing character strings to numbers: not possible. So coded as missing
```

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

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

Missing Data Anyone working with empirical data sooner or later deals with a data set that has missing values. R treats missing values by using a special NA value. You should encode missing data in R as NA and convert any data imports with missing data in other forms to NA as well.

```
missingData<-c(1,3,1,NA,2,1)
missingData
```

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

If computations are performed on data objects with NA values the NA value is carried through to the result.

```
missingData2<-missingData*2
missingData2
```

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

If you have a computation problem with an element of a data object and are not sure whether that is a missing value, the function is.na can be used to determine if the element in question is a NA value.

```
is.na(missingData[1])
```

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

```
is.na(missingData[4])
```

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

Import and Export data sets

We can save the data matrix in an ASCII text file (*.txt) using the write.table() function.

```
write.table(gendat,file="gendat.txt") # Save the matrix in a .txt file
```

You can see the .txt file on your machine using WordPad:

"Eric" "Peter" "Anna"
"gene1" 1 1.5 1.25
"gene2" 1.35 1.55 1
"gene3" -1.1 -1.5 -1.25
"gene4" -1.2 -1.3 -1

To import data into R, we can use the read.table() function.

```
gendat.copy<-read.table("gendat.txt")
gendat.copy
```

```
##        Eric Peter  Anna
## gene1  1.00  1.50  1.25
## gene2  1.35  1.55  1.00
## gene3 -1.10 -1.50 -1.25
## gene4 -1.20 -1.30 -1.00
```

We can also save and import comma delineated spreadsheet files (*.csv) which are used by most spreadsheet softwares. These are done with write.csv() and read.csv() functions.

```
write.csv(gendat,file="gendat.csv")
read.csv("gendat.csv",row.names=1)  # The first column contains the row names, indicated by row.names=1
```

```
##        Eric Peter  Anna
## gene1  1.00  1.50  1.25
## gene2  1.35  1.55  1.00
## gene3 -1.10 -1.50 -1.25
## gene4 -1.20 -1.30 -1.00
```

For more, see the "R Data import/Export" manual, Chapter 3 of the book "R for Beginners" at cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

Computing on data matrix

We have covered some vectors and matrix operations before. Another useful feature on arrays operations is the **apply** functionality. It allows use to apply computations columnwise or rowwise.

```
apply(gendat,1,mean) # Compute mean on all rows
```

```
##  gene1  gene2  gene3  gene4
##  1.250  1.300 -1.283 -1.167
```

```
apply(gendat,2,mean) # Compute mean on all columns
```

```
##   Eric  Peter   Anna
## 0.0125 0.0625 0.0000
```

It frequently happens that we want to re-order the rows of a matrix according to a certain criterion. For instance, to reorder the matrix gendat according to the row means, it is convenient to store these in a vector and to use the function order.

```
meanexprsval <- apply(gendat,1,mean)      # store row means
o <- order(meanexprsval,decreasing=TRUE)  # order by row means in decreasing order
o           # Display the order of row means
```

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

```
gendat[o,]  # Reorder the matrix in decreasing row means
```

```
##        Eric Peter  Anna
## gene2  1.35  1.55  1.00
## gene1  1.00  1.50  1.25
## gene4 -1.20 -1.30 -1.00
## gene3 -1.10 -1.50 -1.25
```

Data Matrix Versus Data Frame

We have mentioned that a data frame can be thought as a matrix whose columns can be refered to by the "$" operator. For a data frame with numerical variables, the computations on matrix can be applied similarly.

```
gendat1<-data.frame(gendat) # Make a new copy (a data frame) of the data matrix
is.data.frame(gendat1) # This is a data frame
```

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

```
is.matrix(gendat1) # NOT a matrix object in R (still contains info in matrix format, just not an R 'mat
```

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

```
apply(gendat1,1,mean) # Compute mean on all rows, same as before
```

```
##  gene1  gene2  gene3  gene4
##  1.250  1.300 -1.283 -1.167
```

```
apply(gendat1,2,mean) # Compute mean on all columns, same as before
```

```
##   Eric  Peter   Anna
## 0.0125 0.0625 0.0000
```

However, for data frames with other types of variables (e.g., character strings), then those matrix computations may not work. Recall, the data frame is in fact a list of those variables. The operations on numerical matrices will only work on the numerical parts of the data frame. We illustrate this on a data set 'iris' comes with R.

```
str(iris) # Display basic info about the 'iris' data set, which is a data frame with four numerical var
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
head(iris) # Show the first few rows of iris
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
apply(iris,2,mean) # Compute column means not working anymore
```

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

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##           NA           NA           NA           NA           NA
```

```
apply(iris[,1:4],2,mean) # Compute column means working on the numerical part: the first four variables
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.843        3.057        3.758        1.199
```

There are other example data sets already built in R. You can check them using the command

```r
data()
```

Lesson 3: Working with the Golub (1999) data set

The textbook by Krijnen often used examples on analyzing the gene expression data collected by Golub et al. (1999). In this lesson, you should get familiar with the data set which will be used frequently in the this course.

Getting the data

The data set 'golub' is contained in the 'multtest' package, which is part of Bioconductor. If you have not previoulsy installed the Bioconductor (see previous module), please install it by the following commands

```r
source("http://www.bioconductor.org/biocLite.R")
biocLite()
```

You only need to install Bioconductor once on your machine. When we want to use the 'golub' data set, we need to load it to the R session. This will need to be done for each R session.

```r
library(multtest) # Load the 'multtest' package
```

```
## Loading required package: Biobase
## Loading required package: BiocGenerics
## Loading required package: parallel
##
## Attaching package: 'BiocGenerics'
##
## The following objects are masked from 'package:parallel':
##
##     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
##     clusterExport, clusterMap, parApply, parCapply, parLapply,
##     parLapplyLB, parRapply, parSapply, parSapplyLB
##
## The following object is masked from 'package:stats':
##
##     xtabs
##
## The following objects are masked from 'package:base':
##
##     anyDuplicated, append, as.data.frame, as.vector, cbind,
##     colnames, do.call, duplicated, eval, evalq, Filter, Find, get,
##     intersect, is.unsorted, lapply, Map, mapply, match, mget,
##     order, paste, pmax, pmax.int, pmin, pmin.int, Position, rank,
##     rbind, Reduce, rep.int, rownames, sapply, setdiff, sort,
##     table, tapply, union, unique, unlist
##
## Welcome to Bioconductor
##
##     Vignettes contain introductory material; view with
##     'browseVignettes()'. To cite Bioconductor, see
##     'citation("Biobase")', and for packages 'citation("pkgname")'.
```

```r
data(golub)       # Load the 'golub' data set
```

The data consist of gene expression values of 3051 genes (rows) from 38 leukemia patients. Twenty seven patients are diagnosed as acute lymphoblastic leukemia (ALL) and eleven as acute myeloid leukemia (AML). The tumor class is given by the numeric vector golub.cl, where ALL is indicated by 0 and AML by 1. The gene names are collected in the matrix **golub.gnames** of which the columns correspond to the gene index, ID, and Name, respectively. We shall first concentrate on expression values of a gene with manufacturer name "M92287_at", which is known in biology as "CCND3 Cyclin D3". The expression values of this gene are collected in row 1042 of golub. To load the data and to obtain relevant information from row 1042 of **golub.gnames**, use the following command.

```r
golub.gnames[1042,]
```

```
## [1] "2354"            "CCND3 Cyclin D3" "M92287_at"
```

The data are stored in a matrix called **golub**. We can see its basic information with the 'str' command.

```r
str(golub) # golub is a 3051 by 38 matrix, with no row names nor column names
```

```
##  num [1:3051, 1:38] -1.458 -0.752 0.457 3.135 2.766 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : NULL
```

The data are pre-processed by procedures described in Dudoit et al. (2002). The matrix has 3051 rows and 38 columns, each data element has a row and a column index. Recall that the first index refers to rows and the second to columns. Hence, the second value from row 1042 is

```r
golub[1042,2]
```

```
## [1] 1.524
```

So 1.524 is the expression value of gene CCND3 Cyclin D3 from patient number 2. We see the expression values of gene CCND3 Cyclin D3 (row 1042) as

```r
golub[1042,]
```

```
##  [1]  2.1089  1.5240  1.9640  2.3360  1.8511  1.9939  2.0660  1.8165
##  [9]  2.1762  1.8086  2.4456  1.9050  2.7661  1.3255  2.5939  1.9278
## [17]  1.1055  1.2765  1.8305  1.7835  0.4583  2.1812  2.3143  1.9993
## [25]  1.3684  2.3735  1.8349  0.8894  1.4501  0.4290  0.8267  0.6364
## [33]  1.0225  0.1276 -0.7433  0.7378  0.4947  1.1206
```

If we want gene CCND3 Cyclin D3 expression values only for the ALL patients, we have to refer to the first twenty seven elements of row 1042.

```r
golub[1042,1:27]
```

However, for the work ahead it is much more convenient to construct a factor indicating the tumor class of the patients. The factor will be called **gol.fac** and is constructed from the vector **golub.cl**, as follows.

```r
gol.fac <- factor(golub.cl, levels=0:1, labels = c("ALL", "AML"))
```

In the sequel this factor will be used frequently. Obviously, the labels correspond to the two tumor classes. The evaluation of gol.fac==“ALL” returns TRUE for the first twenty seven values and FALSE for the remaining eleven. This is useful as a column index for selecting the expression values of the ALL patients.

```r
golub[1042,gol.fac=="ALL"]
```

```
##  [1] 2.1089 1.5240 1.9640 2.3360 1.8511 1.9939 2.0660 1.8165 2.1762 1.8086
## [11] 2.4456 1.9050 2.7661 1.3255 2.5939 1.9278 1.1055 1.2765 1.8305 1.7835
## [21] 0.4583 2.1812 2.3143 1.9993 1.3684 2.3735 1.8349
```

For many types of computations it is very useful to combine a factor with the apply functionality. For instance, to compute the mean gene expression over the ALL patients for each of the genes, we may use the following.

```r
meanALL <- apply(golub[,gol.fac=="ALL"], 1, mean)
```

The specification golub[,gol.fac==“ALL”] selects the matrix with gene expressions corresponding to the ALL patients. The 3051 means are assigned to the vector meanALL.

After reading the classical article by Golub et al. (1999), which is strongly recommended, one becomes easily interested in the properties of certain genes. For instance, gene CD33 plays an important role in distinguishing lymphoid from myeloid lineage cells. To perform computations on the expressions of this gene we need to know its row index. This can obtained by the **grep** function.

```r
grep("CD33",golub.gnames[,2])
```

```
## [1] 808
```

Hence, we can get the expression values of antigen CD33 at golub[808,], and can get further information on it at golub.gnames[808,].

Lesson 4: Displaying data with graphics in R

Data exploration and graphing is the first step in model building. It permits to conduct basic checks and provides quick feel of the patterns in your data. In this section, we will learn how to use R to make plots.

How to choose the charts/Reports form

The two basic divisions of data are categorical (qualitative) data and continuous (quantitative) data. Categorical variables represent types of data which may be divided into groups. Examples of categorical variables are race, sex, age group, and educational level. Continuous variables represent types of the data that is a (real) number.

The choice of charts/reports depends on whether the variables are continuous or categorical.

- Categorical or continuous

Barplots is the usual means to explore categorical variables, while **histograms** and **densitplots** are used for continous variables

- Continuous by continuous.

The relationship between two continuous variables is explored with a **scatterplot**. A scatterplot is sometimes enhanced with the inclusion of a third, categorical, variable using color and/or different symbols

- Continuous by categorical.

Boxplots is often used for examining a continuous variable against a categorical variable

- Categorical by categorical.

A **frequency table** is the usual means of display when examining the relationship between two categorical variables

Bar Charts

Bar charts is used to explore a categorical variable. It can be obtained by the `barplot()` function and the pie chart can be done by the `pie()` function. For example

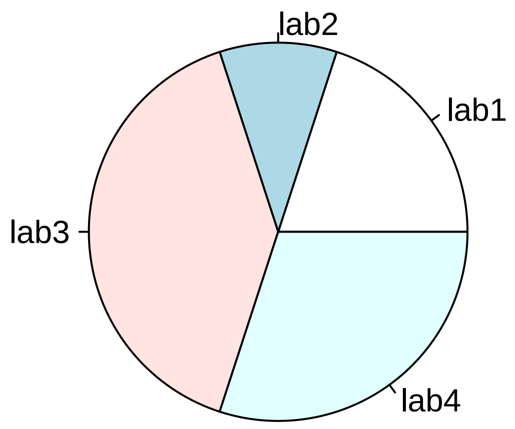
```
labels=c("lab1","lab2","lab3","lab4") # Four categories
x=sample(labels,10,replace=T) # Create a data set with 10 observations: categorical values
x # Display the data set
```

```
##  [1] "lab3" "lab1" "lab1" "lab2" "lab4" "lab3" "lab4" "lab3" "lab3" "lab4"
```

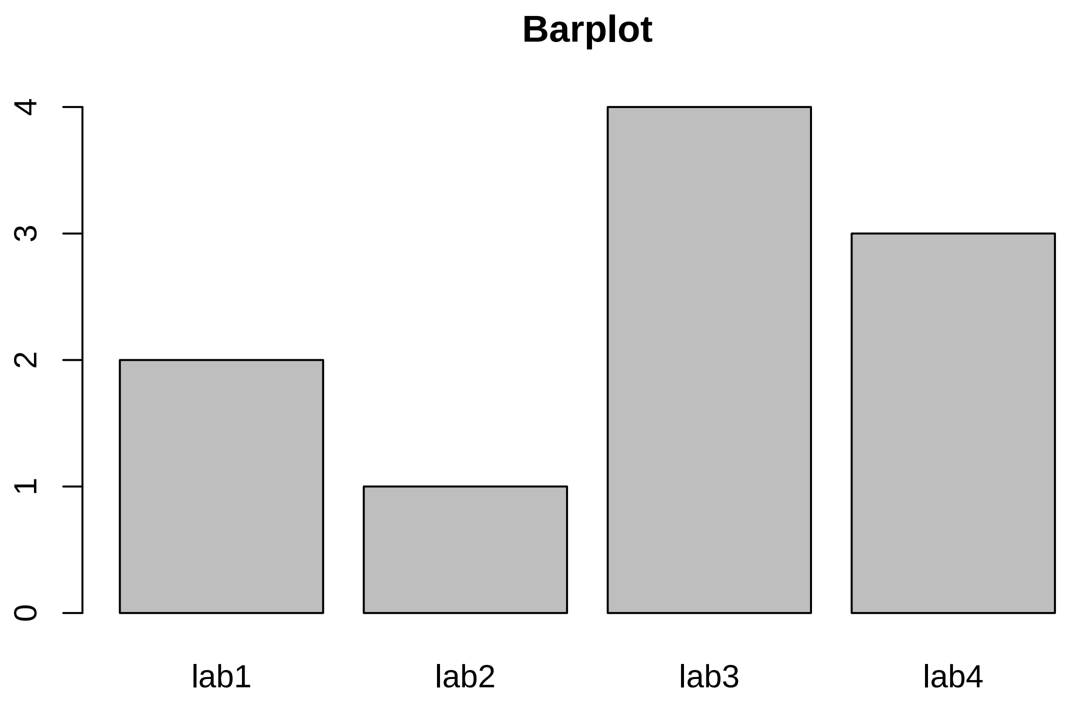
```
table(x)       # frequence table for the data set
```

```
## x
## lab1 lab2 lab3 lab4
##    2    1    4    3
```

```
pie(table(x))  # pie chart for the data set
```



```
barplot(table(x),main="Barplot") # barplot for the data set
```



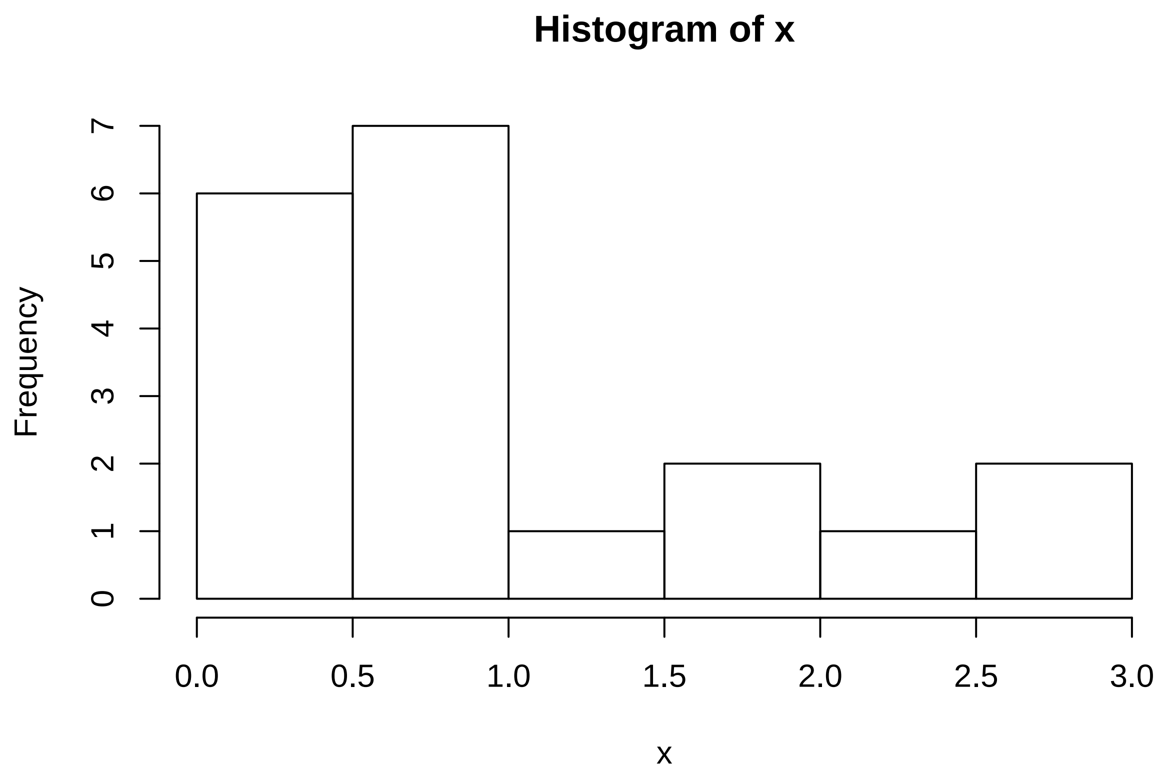
Histograms

Bar charts is used to explore a continuous variable. To make a histogram, we could easily apply the `hist()` function.

```
x <- c((1:10)/10, (1:9)/3)
x  # print x values
```

```
##  [1] 0.1000 0.2000 0.3000 0.4000 0.5000 0.6000 0.7000 0.8000 0.9000 1.0000
## [11] 0.3333 0.6667 1.0000 1.3333 1.6667 2.0000 2.3333 2.6667 3.0000
```

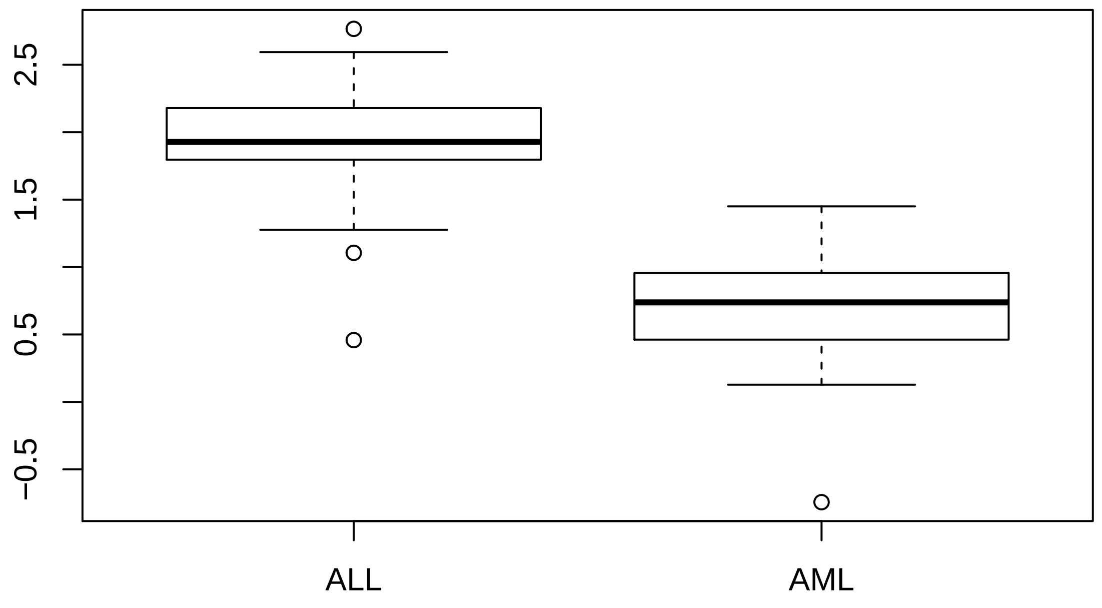
```
hist(x) # draw histogram
```



Boxplot

Boxplots is used for examining a continuous variable against a categorical variable. We can view the distributions of the expression values of the ALL and the AML patients on gene CCND3 Cyclin D3 on two boxplots adjacent to one another.

```r
boxplot(golub[1042,] ~ gol.fac)
```



Frequency table

Frequency table is used to display when examining the relationship between two categorical variables. For example, we randomly generate two categorical variables `sex` and `smoke history`.

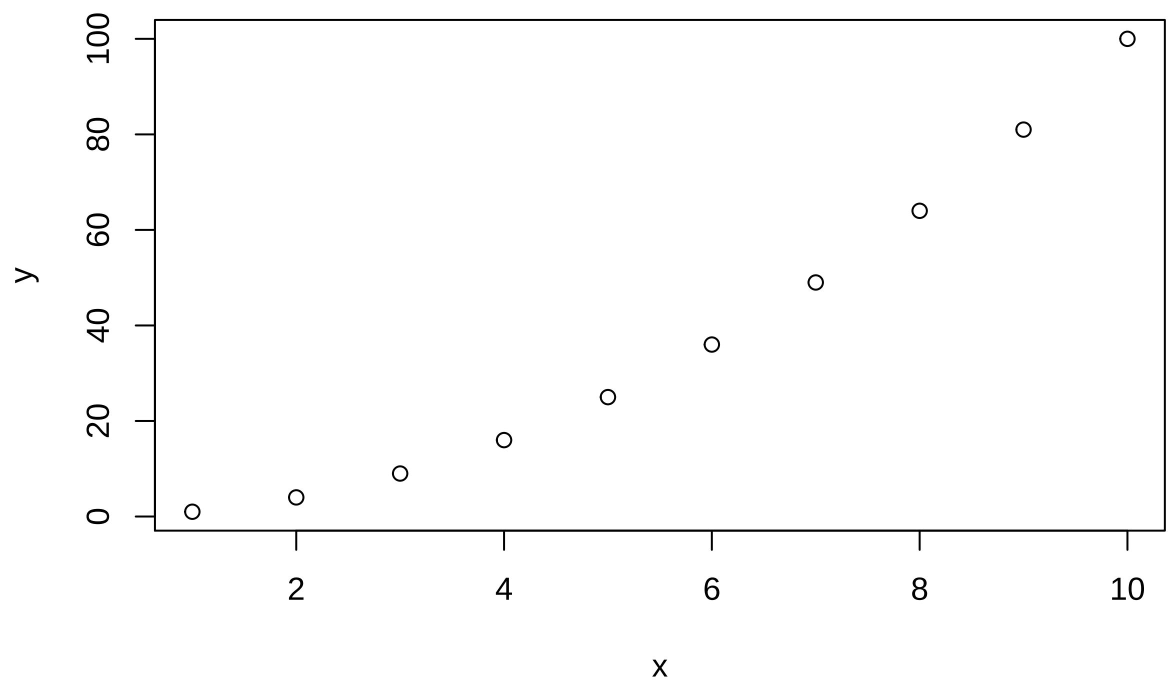
```
sex=sample(c("Male","Female"),10,replace=T)
smoke=sample(c("Smoke","Unsmoke"),10,replace=T)
table(sex,smoke)
```

```
##         smoke
## sex      Smoke Unsmoke
##   Female     3       2
##   Male       4       1
```

Scatter Plot

Scatter plot is used to explore two continuous variables. It is produced by **plot()** function.

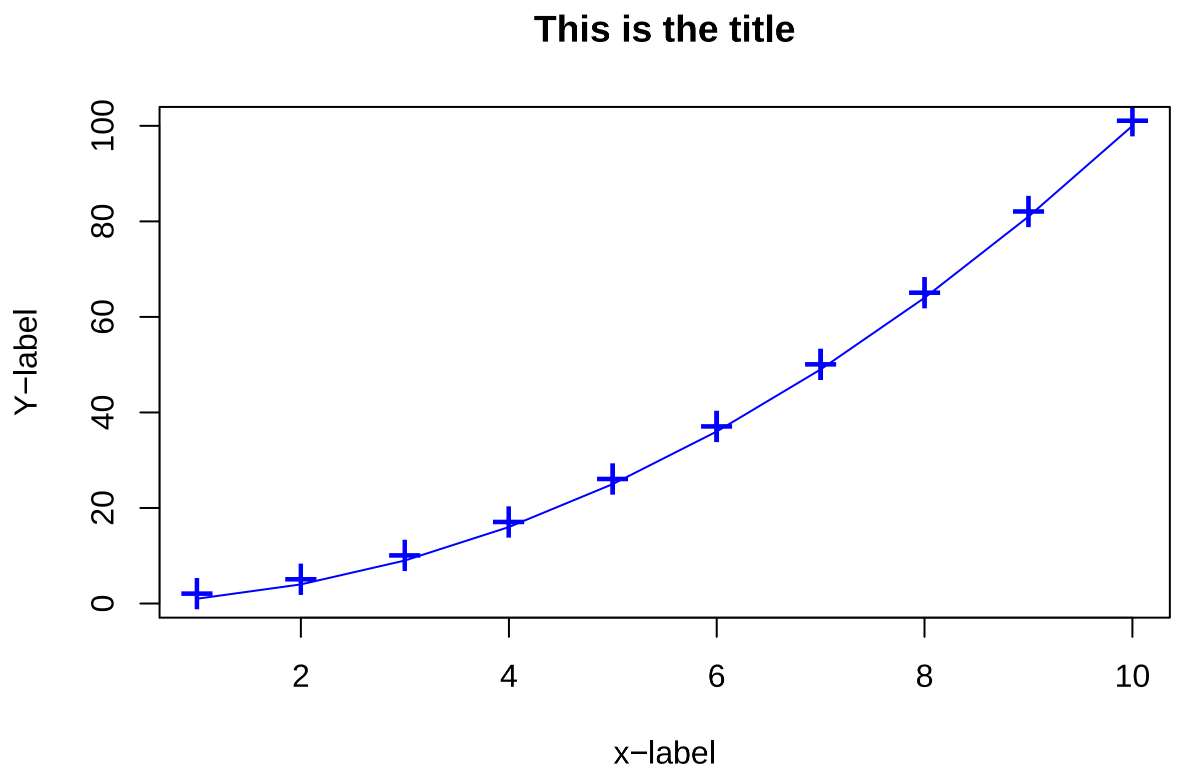
```
x <- 1:10
y <- x^2
plot(x,y) # Same as plot(y~x)
```



Options for plot control

We have shown the basic commands for plotting above. There are more control options in those functions to ajdust graphs to your needs. For example, we could add some more elements, like labels and titles, etc.

```
plot(x,y,xlab="x-label",ylab="Y-label", main="This is the title", type="o", col="blue",pch="+",cex=2)
```

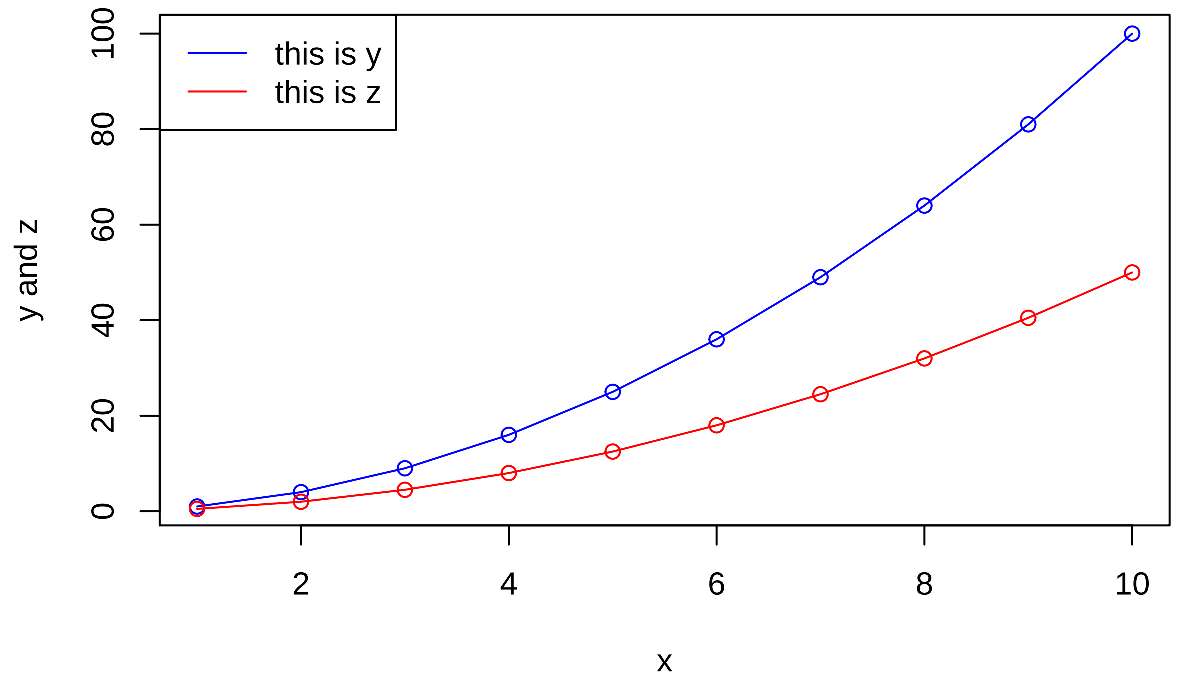


As we can see from the above example, *xlab*,*ylab* and *main* arguments control the labels for x-axis, y-axis, and the title, *type* controls the plotting type, *col* controls the color, *pch* controls the point type, and *cex* controls the size of the point.

For each **plot()** function, it will open a new plot window. We may want to overlap two plots together for comparison. To do this, you can add to an existing plot with **lines()** and **points()** function.

In the following example, we plot y and z against x in the same plot.

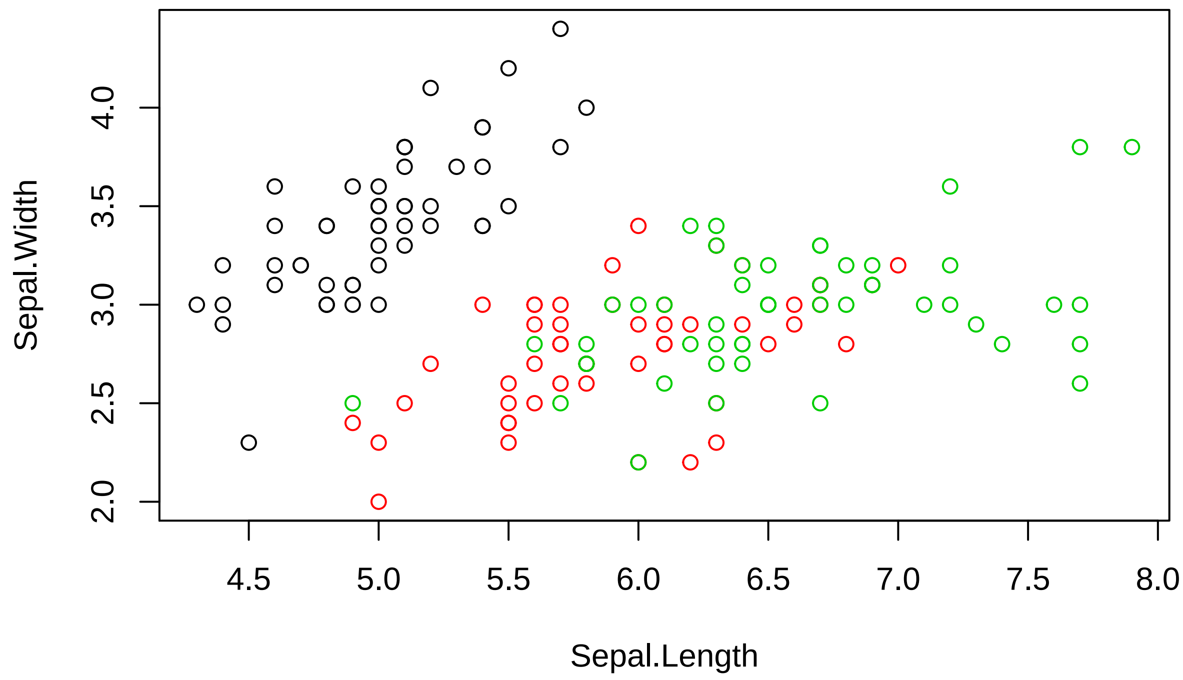
```r
z <- y/2
plot(y~x,col="blue",type="o",ylab="y and z")
lines(x,z,col="red",type="o")
legend("topleft",c("this is y","this is z"),col=c("blue","red"),lty=c(1,1))
```



Plot on data frame

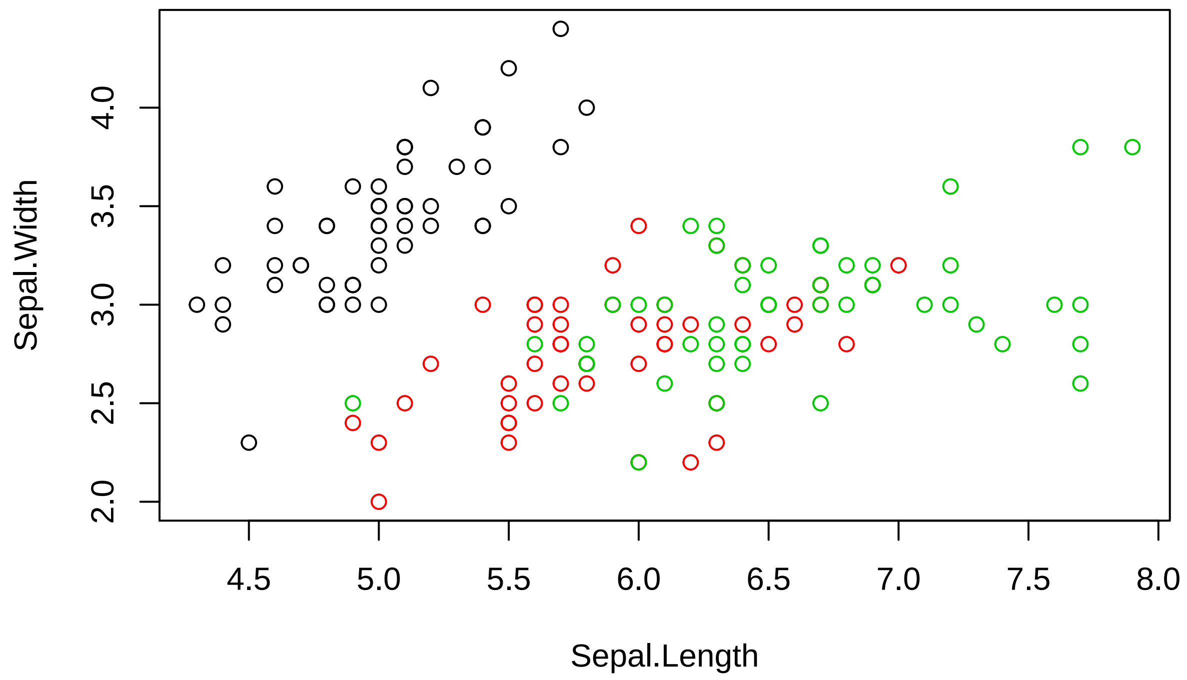
To refer to variables in a data frame, we use '$' operator. In the plot() function, we can also refer to the variables directly by specifying the data frame with the 'data=' option. We illustrate this on the 'iris' data set contained in R. We produce a scatter plot of Sepal Width versus Sepal Length, and label the data points according to the Species.

```
# Referring to the variables by iris$ may be too repetitive. The command
#      plot(iris$Sepal.Width ~ iris$Sepal.Length, col=iris$Species)
# produces the same plot as the following command
plot(Sepal.Width ~ Sepal.Length, col=Species, data=iris)
```

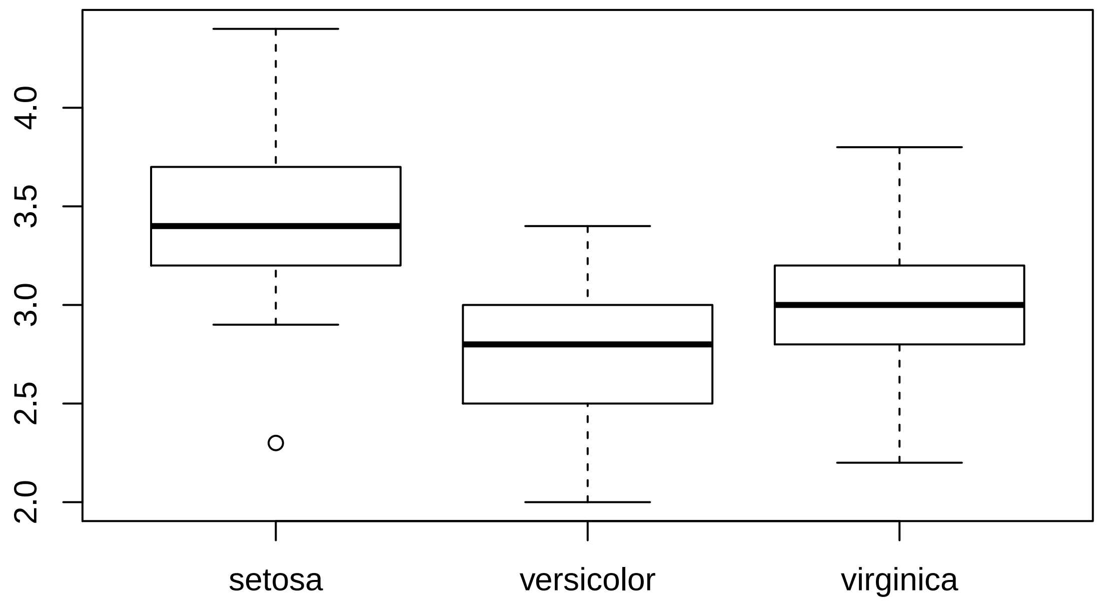


Another alternative is to use with() to specify the data frame.

```
with(iris, {
     plot(Sepal.Width ~ Sepal.Length, col=Species)
     boxplot(Sepal.Width ~ Species)
     }
)
```



Sepal.Width
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Sepal.Length



Saving Graphs

See page 56 of Seefeld & Linder's book.

Finding more help on R

We have covered many basic commands in R. We will introduce more as we go through the course. However, this is a statistics course, not a purely R language course. We will not be able to cover every little details in R. So it is important for you to learn more on your own. One useful feature of R is that it has a help page for every function. You can use ?command to reach the help page of 'command'. For example,

```
?plot
```

should open the help page on 'plot', where you can find details on the syntax of 'plot' inlcuding examples. You can also use the 'Help' on the menu atop your window to reach online (html) help pages, and search for commands.

The website http://www.r-project.org/ also provides online manuals and books about R. There are also website provide online forums and resources on R, such as http://stackoverflow.com/ and http://www.r-bloggers.com/, just to name two.

You can also google helps on any issues related to R. For example, I want to know how to do a plot without the axes produced by default. A google search 'r plot no axis' would results in many links for help documents and answers to similar questions by other persons.

Module summary

We covered several basic types of data objects. We taught how to manipulate these objects, converting the variable types. We covered the basic import/export of data sets to ASCII files and spreadsheets. The data sets are most often stored as data matrices and data frames. You should have learned how to operate on these data sets. Particularly, you should be able to work on data sets already built in R and the data sets in the packages from Bioconductor. We have taught a few basic graphical display of data. Finally, we talked about how to seek further help on R.