Introduction to R

Tatjana Kecojević

2016-11-30

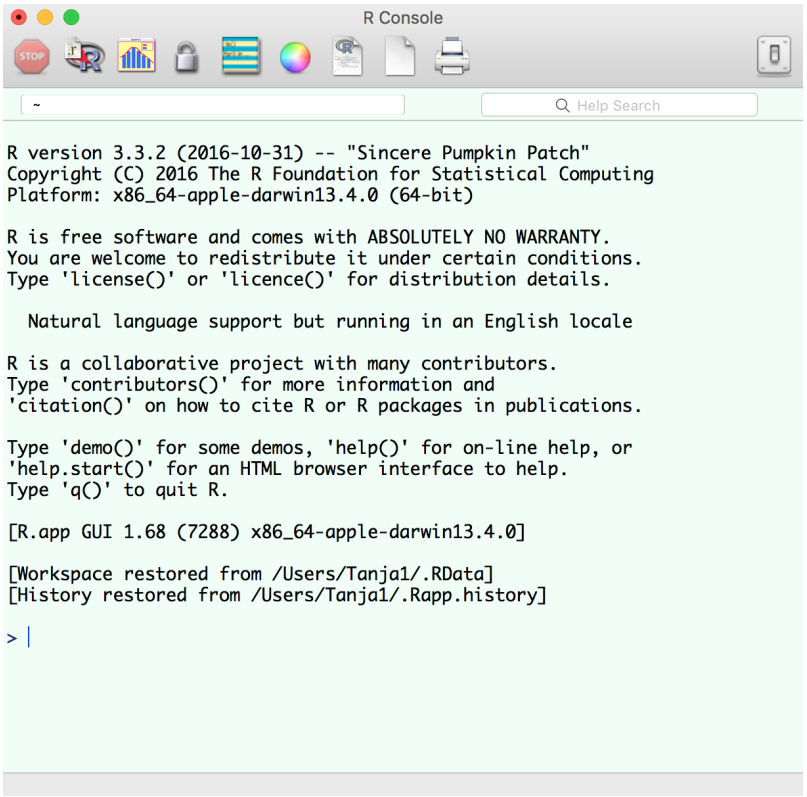
R

The purpose of this section is to provide a basic overview of and introduction to R language and its environment for statistical computing and graphics. R is a public domain language for data analysis, fast becoming the lingua franca of quantitative research with some 9220 free specialised packages. R is a free, open-source data analysis package available for Windows, Mac OS X, and Unix/Linux systems, developed and maintained by R Development Core Team. You can download R from: http://cran.r-project.org.

R can also be run using RStudio which is an open-source integrated development environment (IDE) for R, which means it allows the user to run R in a more user-friendly environment. You can download RStudio from: https://www.rstudio.com/products/rstudio/download/

Getting Started

To run R you need to click on the R icon on your computer. When R is launched you will see a single window called R Console



The simplest way to get help in R is to click on the Help option on the toolbar of the R console or RStudio, depending on which one you decide to use. Alternatively, to get help type

```
help.start()
```

which will provide a web-based interface to the help system. There is a vast array of guidebooks on the web that will help you with R, but your first port of call should be CRAN's website http://cran.r-project.org/ where you can find a number of manuals. Visiting CRAN's website is useful as you will be able to search through *Frequently Asked Questions (FAQs)* and *R News* which will keep you up to date with new useful articles, book reviews and dates of forthcoming releases.

To begin with, we can use R as a calculator:

```
5+9
```

```
## [1] 14
```

```
3-2
```

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

```
18/6
```

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

```
4*7
```

```
## [1] 28
```

```
(5-3)^2/4
```

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

```
9^(1/2)*4
```

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

Note that you don't have to type the equals sighn and that each answer has **[1]** in front. The **[1]** indicates that there is only one number in the answer. If the answer contains more than one number it uses numbering like this to indicate where in the 'group' of numbers each one is.

R provides a number of specialised data structures we will refer to as **objects**. To refer to an object we use a symbol. You can assign any object using the assignment operator **<-**, which is a composite made up from 'less than' and 'minus', with *no space between them!* Thus, we can create scalar constants, which we refer to as variables, and perform mathematical operations over them.

```
x <- 5
y <- 6
```

You can use objects in calculation in exactly the same way as as you have already seen numbers being used earlier:

```
x+y
```

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

and you can store the results of the calculation done with the objects in another object:

```
z <- x*y
z
```

```
## [1] 30
```

BUT, remember!!! Operator **<-** is a composite made up from 'less than' and 'minus', with no space between them!!!! Try to type:

```
x< -5
y< -6
```

and see what happens.

After you've created some objects in R you can get a list of them using **ls()** function:

```
ls()
```

```
## [1] "x" "y" "z"
```

You can also remove an object from R's 'workspace' using **rm()** function.

```
rm(z)
ls()
```

```
## [1] "x" "y"
```

R is not like other conventional statistical packages like SAS, Minitab, SPSS, to name a few. It is more of a programming language designed for conducting data analyses. It comes with a vast number of ready-made blocks of code that will enable you to manipulate data, perform intricate mathematical calculations with data, carry out an array of statistical analysis ranging from simple to complex to extremely complex and it will facilitate the creation of fantastic graphs. These pre-made blocks of code are known as **functions**.

R has all the standard mathematical functions that you might ever need: *sin, cos, tan, asin, atan, log, log10, exp, abs, sqrt, factorial...* To use them, all you need to do is to type the function and put the name of the object (argument) you would like to use the function for in brackets.

```
sqrt(144)
```

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

```
log10(8)
```

```
## [1] 0.90309
```

```
log10(100)
```

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

```
log(100)
```

```
## [1] 4.60517
```

```
exp(1)
```

```
## [1] 2.718282
```

```
pi
```

```
## [1] 3.141593
```

```
sin(pi/2)
```

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

```
abs(-7)
```

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

```
factorial(3)
```

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

```
exp(x)
```

```
## [1] 148.4132
```

```
log(y, 2)
```

```
## [1] 2.584963
```

You can use expression as argument of a function:

```
z <- x*y
trunc(x^2 + z/y)
```

```
## [1] 30
```

```
log((100*x - y^2)/z)
```

```
## [1] 2.738687
```

You can have nested functions and you can use functions in creating new objects:

```
round(exp(x), 2)
```

```
## [1] 148.41
```

```
p <- abs(floor(log((100*x - y^2) / exp(z))))
p
```

```
## [1] 24
```

Vectors and Matrices

When analysed data you are more likely to be working with lots of numbers/variables. It would be much more convenient to keep all of those number/variables as an object. Variables can be of a different type: logical, integer, double, string are some examples. Variables with one or more values *of the same type* are **vectors**. Hence, a variable with a single value (known to us as a scalar) is a vector of length 1. We can assign to vectors in many different ways:

- generated by R using the colon symbol (**:**) as a sequence generated operator or by using built in function `rep` for replicating the given number for a given number of times

```
x <- 1:10
x
```

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

```
x <- rep(1,10)
x
```

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

- generated by the user by using concatenation function **c** or using function **scan** that allows you to enter one number at a the time, When using scan to indicate that the input is complete you need to press the Enter button.

```
x <- c(2, 6, 4, 2, 3, 7, 1, 5, 9, 8)
x
```

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

```
y <- scan()
y
```

```
## numeric(0)
```

- created as a sequence of numbers. For example to generate a sequence of numbers from 1 to 10, with increments of 0.2 type

```
seq(1,10,0.2)
```

```
##  [1]  1.0  1.2  1.4  1.6  1.8  2.0  2.2  2.4  2.6  2.8  3.0  3.2  3.4  3.6
## [15]  3.8  4.0  4.2  4.4  4.6  4.8  5.0  5.2  5.4  5.6  5.8  6.0  6.2  6.4
## [29]  6.6  6.8  7.0  7.2  7.4  7.6  7.8  8.0  8.2  8.4  8.6  8.8  9.0  9.2
## [43]  9.4  9.6  9.8 10.0
```

R can easily perform arithmetic with vectors as it does with scalars. You have already seen that R contains a number of operators. There is a list of some of them you are likely to use the most:

- arithmetic: +, -, *, , %%
- relational: >, >=, <=, ==, !=
- logical: !, &, |

Thus, just as we can use those operators over the scalars we can use them when dealing with the vectors and/or a combination of both.

```
x <- rep(1,10)
y <- 1:10
x
```

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

```
y
```

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

```
c(x, y)
```

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

```
x+y
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
x+2*y
```

```
##  [1]  3  5  7  9 11 13 15 17 19 21
```

```
x^2/y
```

```
##  [1] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000 0.1666667 0.1428571
##  [8] 0.1250000 0.1111111 0.1000000
```

```
z <- (x+y)/2
z
```

```
##  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

```
z <- c(z, rep(1, 3), c(100, 200, 300))+1
z
```

```
##  [1]   2.0   2.5   3.0   3.5   4.0   4.5   5.0   5.5   6.0   6.5   2.0
## [12]   2.0   2.0 101.0 201.0 301.0
```

```
p <- 2.5
z*p
```

```
##  [1]   5.00   6.25   7.50   8.75  10.00  11.25  12.50  13.75  15.00  16.25
## [11]   5.00   5.00   5.00 252.50 502.50 752.50
```

It can get messy with all the objects you create. To keep your house in order it is useful to check from time to time what is there. Remember, to list all the objects you have created, use function ls( ):

```
ls()
```

```
## [1] "p" "x" "y" "z"
```

To access a specific element of a vector you would use index inside a single square bracket [] operator. The following shows how to obtain a vector member. The vector index is 1-based, thus use the index position 4 to access the fourth element.

```
y <- c(9, 3, 7, 2, 9, 2, 1, 5, 4, 6)
y
```

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

Note that missing values in R are represented by the symbol **NA** (*not available*() or **NaN** (*not a number*) for undefined mathematical operations. Here, `NA` would be shown if an index is out-of-range.

You can also obtain a desirable selection of the elements of a vector by specifying a query within the index brackets []:

```
y
```

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

```
y[y>5]
```

```
## [1] 9 7 9 6
```

```
y[y>10]
```

```
## numeric(0)
```

```
y[y!=2]
```

```
## [1] 9 3 7 9 1 5 4 6
```

In R you can evaluate functions over the entire vectors which helps to avoid from looping.

```
max(y)
```

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

```
range(y)
```

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

```
mean(y)
```

```
## [1] 4.8
```

```
var(y)
```

```
## [1] 8.4
```

```
sort(y)
```

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

```
cumsum(y)
```

```
##  [1]  9 12 19 21 30 32 33 38 42 48
```

To obtain a description of a function you need to type a question mark, **?**, in front of the name of the function. You might find this particularly useful when you start applying more complicated functions, as help will often provide you not only with the detailed description of the function's input/output arguments, but practical illustrative examples on how the function can be used and applied.

```
?mean
```

When data is arranged in two dimensions rather than one we have **matrices**. In R function **matrix( )** creates matrices:

```
ma1 <- matrix(c(1, 0, -20, 0, 1, -15, 1, -1, 0), nrow=3,
ncol=3, byrow=T)
ma1
```

```
##      [,1] [,2] [,3]
## [1,]    1    0  -20
## [2,]    0    1  -15
## [3,]    1   -1    0
```

The individual numbers in a matrix are called the *elements* of the matrix. Each element is uniquely defined by its particular *row number* and *column number*. To determine the dimensions of a matrix use function **dim( )**

```
dim(ma1)
```

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

An element at the i^{th} row, j^{th} column of a matrix can be accessed by indexing inside square bracket operator **[i, j]**.

```
ma1[2,3]
```

```
## [1] -15
```

The entire i^{th} row or entire j^{th} column of a matrix can be extracted as

```
ma1[3, ]
```

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

```
ma1[ ,2]
```

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

Standard scalar algebra, which deals with operations on single numbers, has a set of well established rules for handling manipulations involving addition, subtraction, multiplication and division. In a broadly similar fashion a set of rules has been developed to enable us to manipulate matrices. However, introducing those rules is beyond the scope of this session and they are covered in the complementary set of notes *Matrix Methods in R*.

Data Types

The examples we have used so far are all dealing with numbers (quantitative numerical data). Those of you familiar with programming will know that **numerical objects** can be classified as *real*, *integer*, *double* or *complex*. To check if an object is numeric and of what type it is, you can use **mode( )** and **class( )** functions respectively.

```
x <- 1:10
mode(x)
```

```
## [1] "numeric"
```

```
class(x)
```

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

In R to enter **strings of characters** as objects you need to enter them using quote marks around them. By default `R` expects all inputs to be numeric and unless you use quote marks around the strings you wish to enter, it will consider them as numbers and subsequently will return an error message.

```r
x <- c("UK", "Spain", "France", "Germany", "Italy")
mode(x)
```

```
## [1] "character"
```

It is common in statistical data to have *attribute* also known as *categorical variables*. In R such variables should be specified as **factors**. Attribute variable has a set of *levels* indicating possible outcomes. Hence, to deal with x as an attribute variable with five levels we need to make it a factor in R.

```r
x <- factor(x)
x
```

```
## [1] UK      Spain   France  Germany Italy
## Levels: France Germany Italy Spain UK
```

Note that R codes the factor levels in their alphabetical order. However, attribute variables are usual coded and you would ususally enter them as such.

```r
quality <- factor(c(3, 3, 4, 2, 2, 4, 0, 1))
levels(quality)
```

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

```r
quality
```

```
## [1] 3 3 4 2 2 4 0 1
## Levels: 0 1 2 3 4
```

You might need to deal from time to time with **logical** data type. This is when something is recorded as *TRUE* or *FALSE*. It is most likely that you would use this data type when checking what type of data the variable is that you are dealing with. For example

```r
is.numeric(x)
```

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

```r
is.factor(x)
```

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

Data Frames

Statistical data usually consists of several vectors of equal length and of various types that resemble a table. Those vectors are interconnected across so that data in the same position comes from the same experimental unit, ie. observation. R uses **data frame** for storing this kind of data table and it is regarded as primary data structure.

Let us consider a study of share prices of companies from three different business sectors. As part of the study a random sample (n=15) of companies was selected and the following data was collected:

```r
share_price <- c(880, 862, 850, 840, 838, 799, 783, 777, 767, 746, 692, 689, 683, 661, 658)
profit <- c(161.3, 170.5, 140.7, 115.7, 107.9, 135.2, 142.7, 114.9, 110.4, 98.9, 90.2, 80.6, 85.4, 91.7
sector<-factor(c(3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1, + 1, 1)) # 1:IT, 2:Finance, 3:Pharmaceutical
#
share_price
```

```
##  [1] 880 862 850 840 838 799 783 777 767 746 692 689 683 661 658
```

```r
profit
```

```
##  [1] 161.3 170.5 140.7 115.7 107.9 135.2 142.7 114.9 110.4  98.9  90.2
## [12]  80.6  85.4  91.7 137.8
```

```
sector
```

```
##  [1] 3 3 3 3 3 2 2 2 2 2 1 1 1 1 1
## Levels: 1 2 3
```

Rather than keeping this data as a set of individual vectors in R, it would be better to keep whole data as a single object, i.e. data frame.

```
share.data<-data.frame(share_price, profit, sector)
share.data
```

```
##    share_price profit sector
## 1          880  161.3      3
## 2          862  170.5      3
## 3          850  140.7      3
## 4          840  115.7      3
## 5          838  107.9      3
## 6          799  135.2      2
## 7          783  142.7      2
## 8          777  114.9      2
## 9          767  110.4      2
## 10         746   98.9      2
## 11         692   90.2      1
## 12         689   80.6      1
## 13         683   85.4      1
## 14         661   91.7      1
## 15         658  137.8      1
```

Individual vectors from the data frame can be accessed using $ symbol:

```
share.data$sector
```

```
##  [1] 3 3 3 3 3 2 2 2 2 2 1 1 1 1 1
## Levels: 1 2 3
```

This notation for accessing variables in data frames can become rather gruelling when having to type it repeatedly. By attaching data frame to R *search path*, variables from that data frame can be accessed by simply giving their names.

```
attach(share.data)
```

```
## The following objects are masked _by_ .GlobalEnv:
##
##     profit, sector, share_price
```

From now on if you type the name of a variable in the attached data frame you do not have to tell R the name of the data frame in which it can be found. You can always check what is in *system's search path* by using **search( )**:

```
search()
```

```
##  [1] ".GlobalEnv"        "share.data"        "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

.GlobalEnv is the *workspace* and *package:base* is the *system library, i.e. package* where all standard functions are defined. The rest of the so called *base packages* contain the basic statistical routines. Assuming that you

are connected to the internet, you can install a package using *install.packages()*. Note that if you are working behind a firewall system, it will try to prevent you from adding new packages. To enable installation of new packages type the following:

```
setInternet2()
chooseCRANmirror()
```

You will be asked to select the *mirror* nearest to you for fast downloading (any UK would be fine), then everything else is automatic.

```
install.packages("ggplot2")
```

Once you have installed a package to use it you have to load it to *system's search path* by simply typing

```
library(ggplot2)
```

Many packages inside and outside the standard R distribution, come with built-in data sets. For example, ggplot contains *economics* data set:

```
data(economics)
economics[1:5,]
```

```
## # A tibble: 5 x 6
##         date   pce    pop psavert uempmed unemploy
##       <date> <dbl>  <int>   <dbl>   <dbl>    <int>
## 1 1967-07-01 507.4 198712    12.5     4.5     2944
## 2 1967-08-01 510.5 198911    12.5     4.7     2945
## 3 1967-09-01 516.3 199113    11.7     4.6     2958
## 4 1967-10-01 512.9 199311    12.5     4.9     3143
## 5 1967-11-01 518.1 199498    12.5     4.7     3066
```

However, often you will have to load data from an out source, such as spreadsheet or database or maybe another statistical package. Loading data into R is not difficult and the key command you will use will be *read.table*. Commonly, data is saved in a text file (for example: *mydata.txt*) and to load this data into R you would type the following:

```
mydata <- read.table("c:/mydata.txt", header=TRUE)
```

Sometimes you will still find it easier to manipulate and organise your data using EXCEL. If this is the case you can save the spreadsheet as a *csv* file (Comma Separated Values file) that can be loaded into R by using read.csv command.

```
mydata <- read.csv("c:/mydata.csv", header=TRUE)
```

Note that when you ask R to load a data file you need to specify the exact path of where the file is saved by giving the full path name in quotes.

Remember to keep your house in order! When you are done with an attached data frame you should remove it from *system's search path* using **detach( )**.

```
detach(share.data)
search()
```

```
##  [1] ".GlobalEnv"        "package:ggplot2"   "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

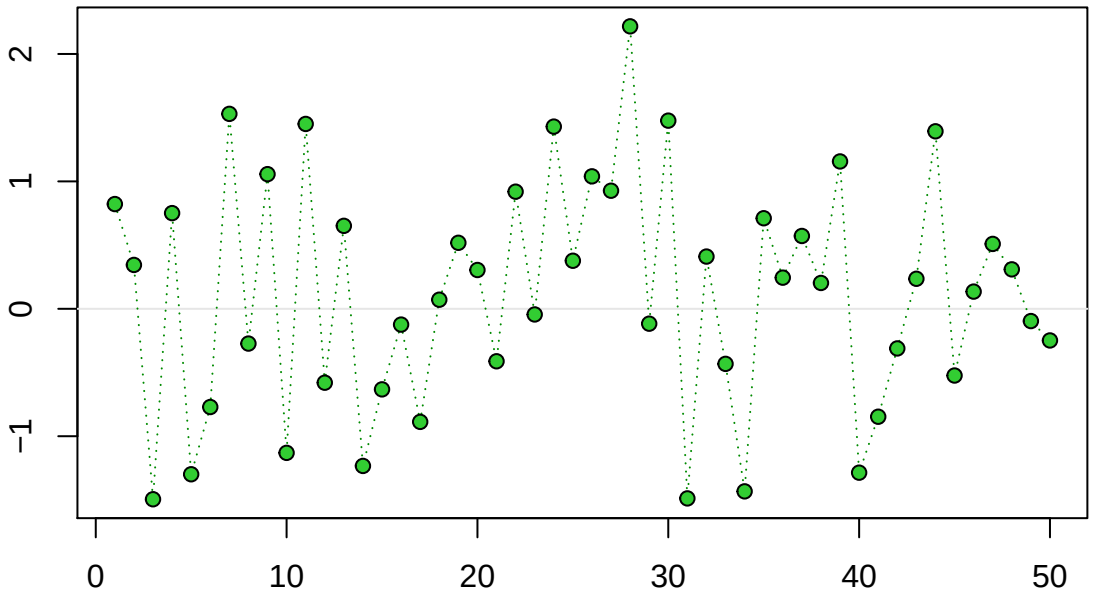
If you are going to quit your R session right away this is not necessary as quitting detaches everything that was attached.

Graphs

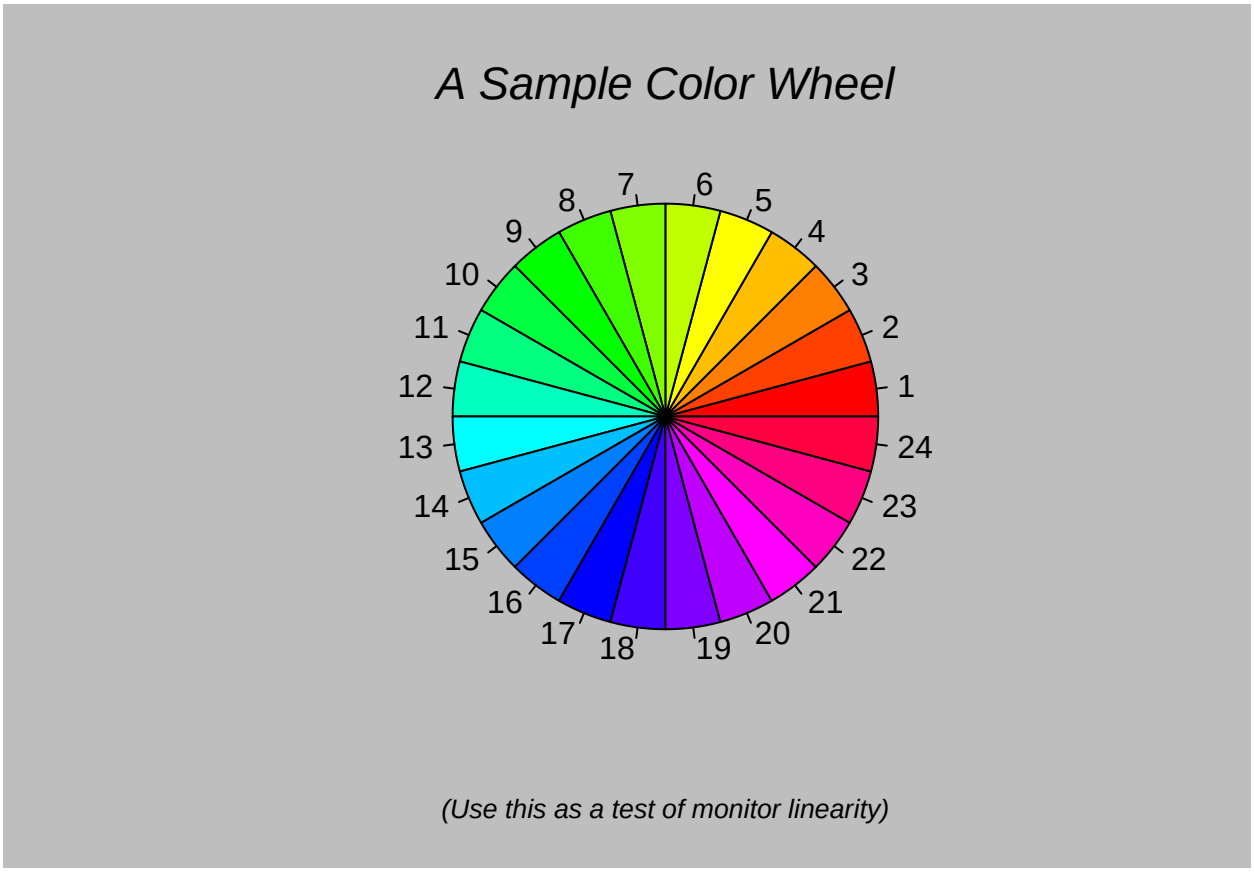
R has many great functions for producing high quality plots. To get the idea of what type of plots it is possible to produce in R type the following:

```
demo(graphics)
```

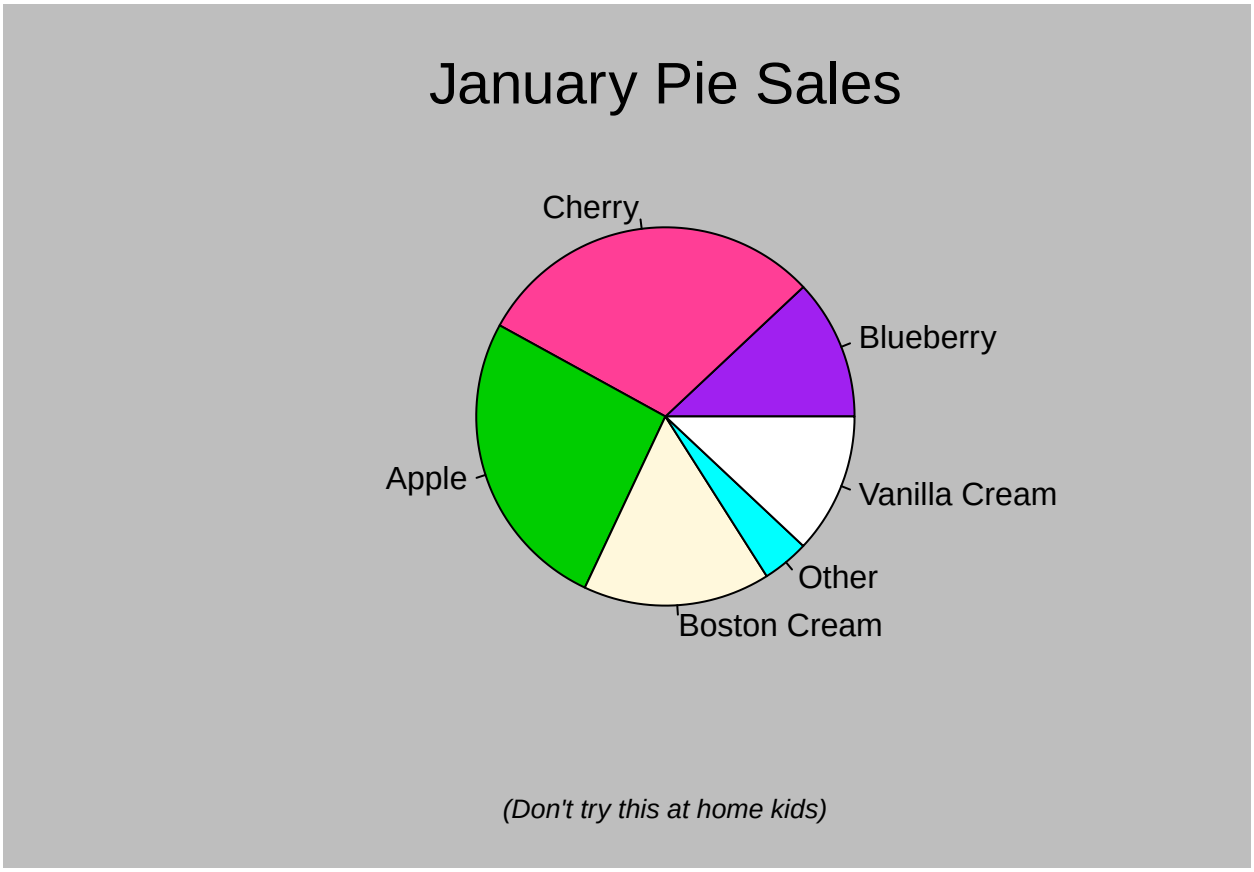
```
##
##
##  demo(graphics)
##  ---- ~~~~~~~~
##
## > #  Copyright (C) 1997-2009 The R Core Team
## >
## > require(datasets)
##
## > require(grDevices); require(graphics)
##
## > ## Here is some code which illustrates some of the differences between
## > ## R and S graphics capabilities.  Note that colors are generally specified
## > ## by a character string name (taken from the X11 rgb.txt file) and that line
## > ## textures are given similarly.  The parameter "bg" sets the background
## > ## parameter for the plot and there is also an "fg" parameter which sets
## > ## the foreground color.
## >
## >
## > x <- stats::rnorm(50)
##
## > opar <- par(bg = "white")
##
## > plot(x, ann = FALSE, type = "n")
```



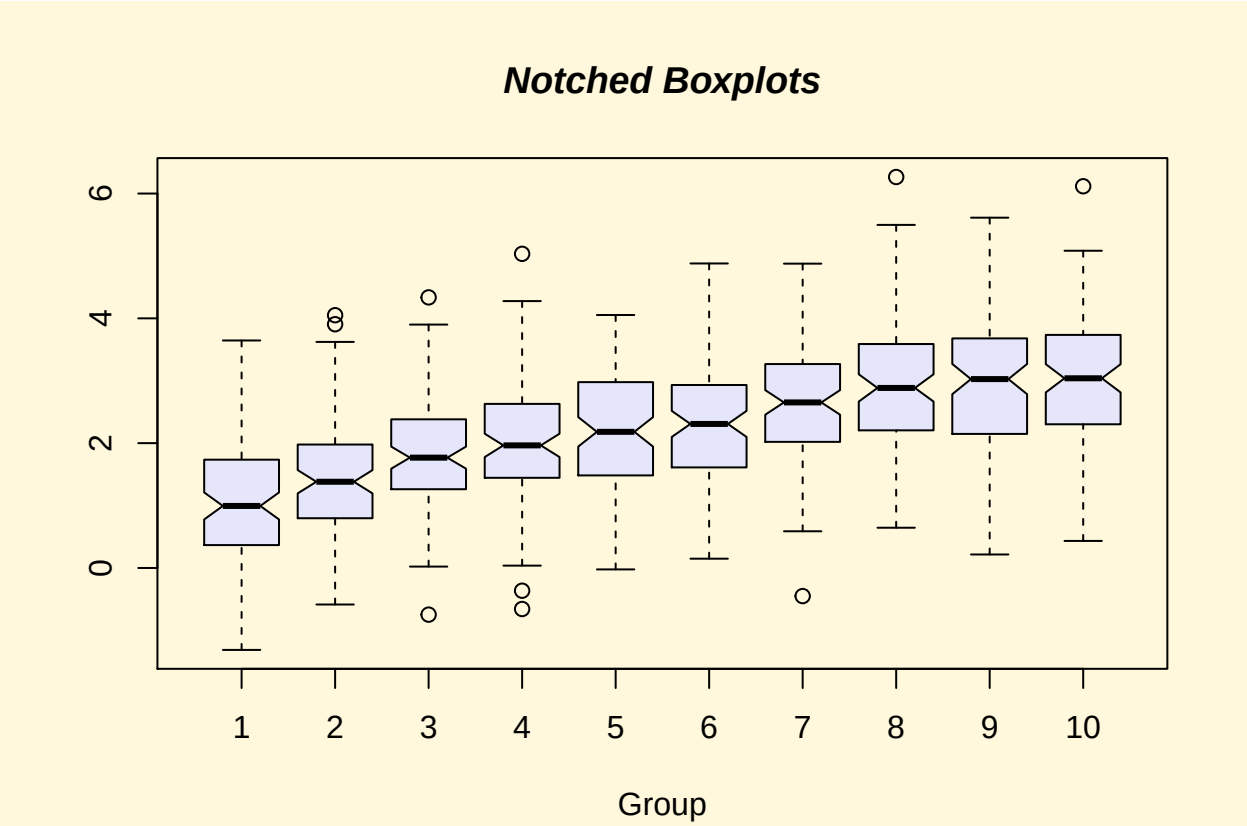

```
##
## > abline(h = 0, col = gray(.90))
##
## > lines(x, col = "green4", lty = "dotted")
##
## > points(x, bg = "limegreen", pch = 21)
##
## > title(main = "Simple Use of Color In a Plot",
## +       xlab = "Just a Whisper of a Label",
## +       col.main = "blue", col.lab = gray(.8),
## +       cex.main = 1.2, cex.lab = 1.0, font.main = 4, font.lab = 3)
##
## > ## A little color wheel.    This code just plots equally spaced hues in
## > ## a pie chart.    If you have a cheap SVGA monitor (like me) you will
## > ## probably find that numerically equispaced does not mean visually
## > ## equispaced.  On my display at home, these colors tend to cluster at
## > ## the RGB primaries.  On the other hand on the SGI Indy at work the
## > ## effect is near perfect.
## >
## > par(bg = "gray")
##
## > pie(rep(1,24), col = rainbow(24), radius = 0.9)
```



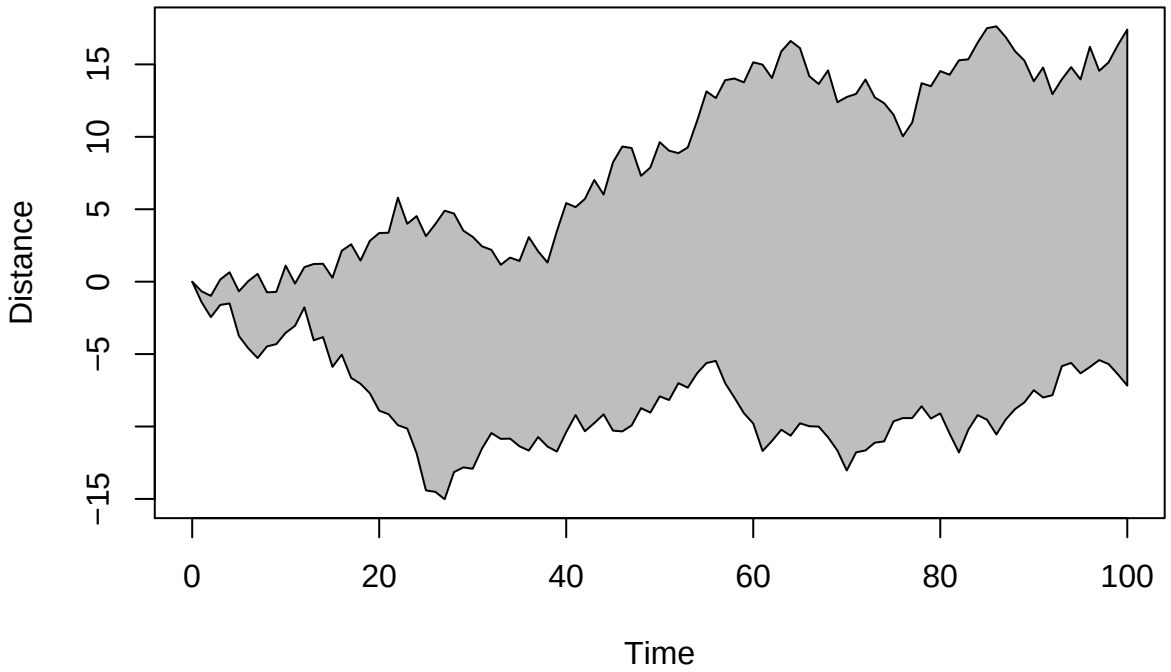

```
##
## > title(main = "A Sample Color Wheel", cex.main = 1.4, font.main = 3)
##
## > title(xlab = "(Use this as a test of monitor linearity)",
## +       cex.lab = 0.8, font.lab = 3)
##
## > ## We have already confessed to having these.  This is just showing off X11
## > ## color names (and the example (from the postscript manual) is pretty "cute".
## >
## > pie.sales <- c(0.12, 0.3, 0.26, 0.16, 0.04, 0.12)
##
## > names(pie.sales) <- c("Blueberry", "Cherry",
## +              "Apple", "Boston Cream", "Other", "Vanilla Cream")
##
## > pie(pie.sales,
## +     col = c("purple","violetred1","green3","cornsilk","cyan","white"))
```



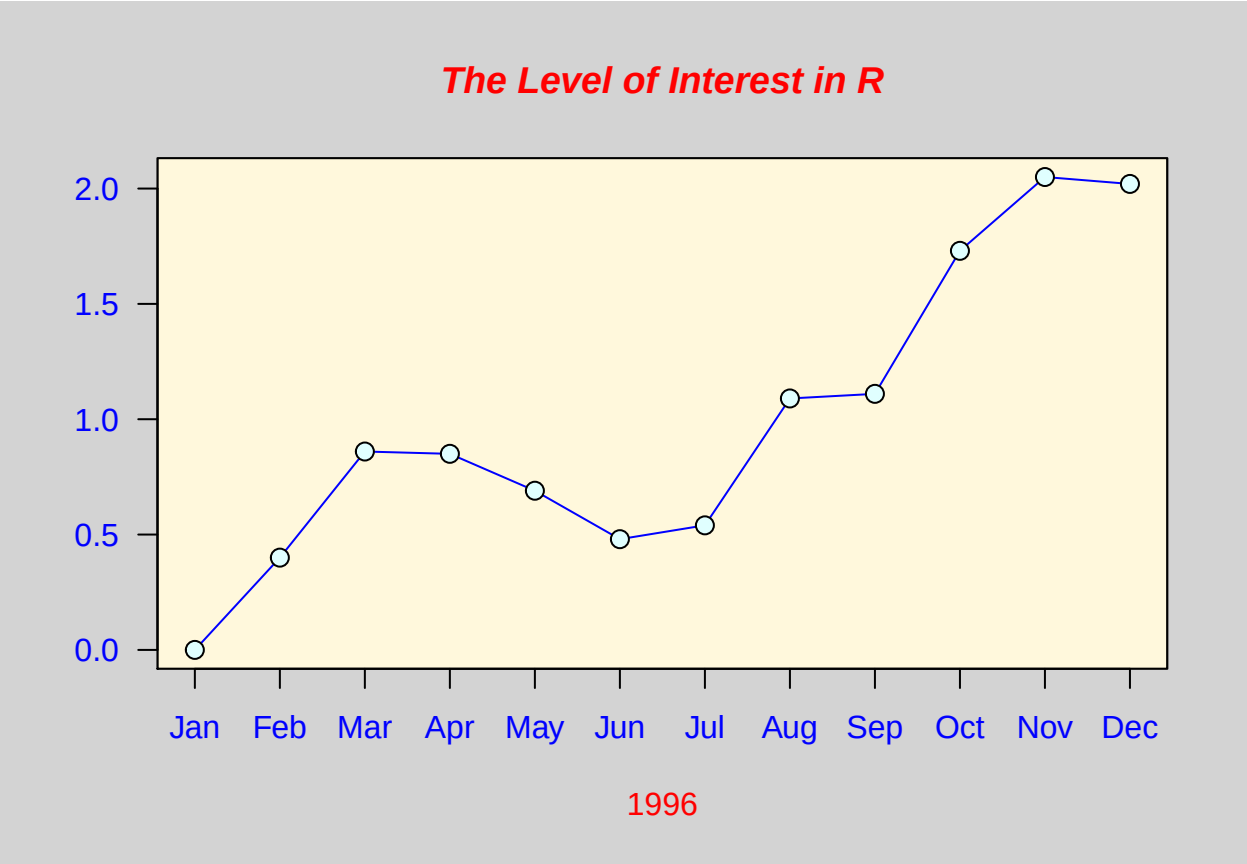

```
##
## > title(main = "January Pie Sales", cex.main = 1.8, font.main = 1)
##
## > title(xlab = "(Don't try this at home kids)", cex.lab = 0.8, font.lab = 3)
##
## > ## Boxplots:  I couldn't resist the capability for filling the "box".
## > ## The use of color seems like a useful addition, it focuses attention
## > ## on the central bulk of the data.
## >
## > par(bg="cornsilk")
##
## > n <- 10
##
## > g <- gl(n, 100, n*100)
##
## > x <- rnorm(n*100) + sqrt(as.numeric(g))
##
## > boxplot(split(x,g), col="lavender", notch=TRUE)
```




```
##
## > title(main="Notched Boxplots", xlab="Group", font.main=4, font.lab=1)
##
## > ## An example showing how to fill between curves.
## >
## > par(bg="white")
##
## > n <- 100
##
## > x <- c(0,cumsum(rnorm(n)))
##
## > y <- c(0,cumsum(rnorm(n)))
##
## > xx <- c(0:n, n:0)
##
## > yy <- c(x, rev(y))
##
## > plot(xx, yy, type="n", xlab="Time", ylab="Distance")
```

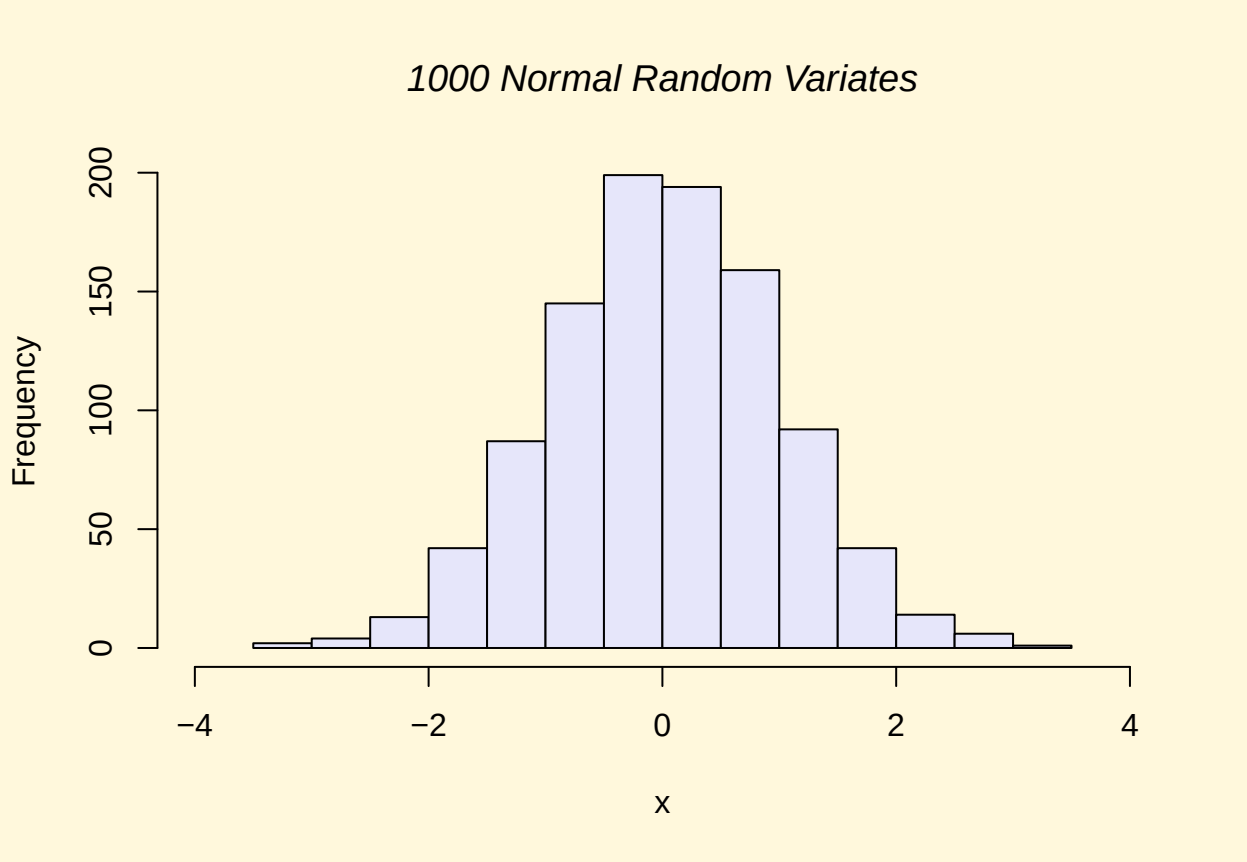



```
##
## > polygon(xx, yy, col="gray")
##
## > title("Distance Between Brownian Motions")
##
## > ## Colored plot margins, axis labels and titles.    You do need to be
## > ## careful with these kinds of effects.    It's easy to go completely
## > ## over the top and you can end up with your lunch all over the keyboard.
## > ## On the other hand, my market research clients love it.
## >
## > x <- c(0.00, 0.40, 0.86, 0.85, 0.69, 0.48, 0.54, 1.09, 1.11, 1.73, 2.05, 2.02)
##
## > par(bg="lightgray")
##
## > plot(x, type="n", axes=FALSE, ann=FALSE)
```

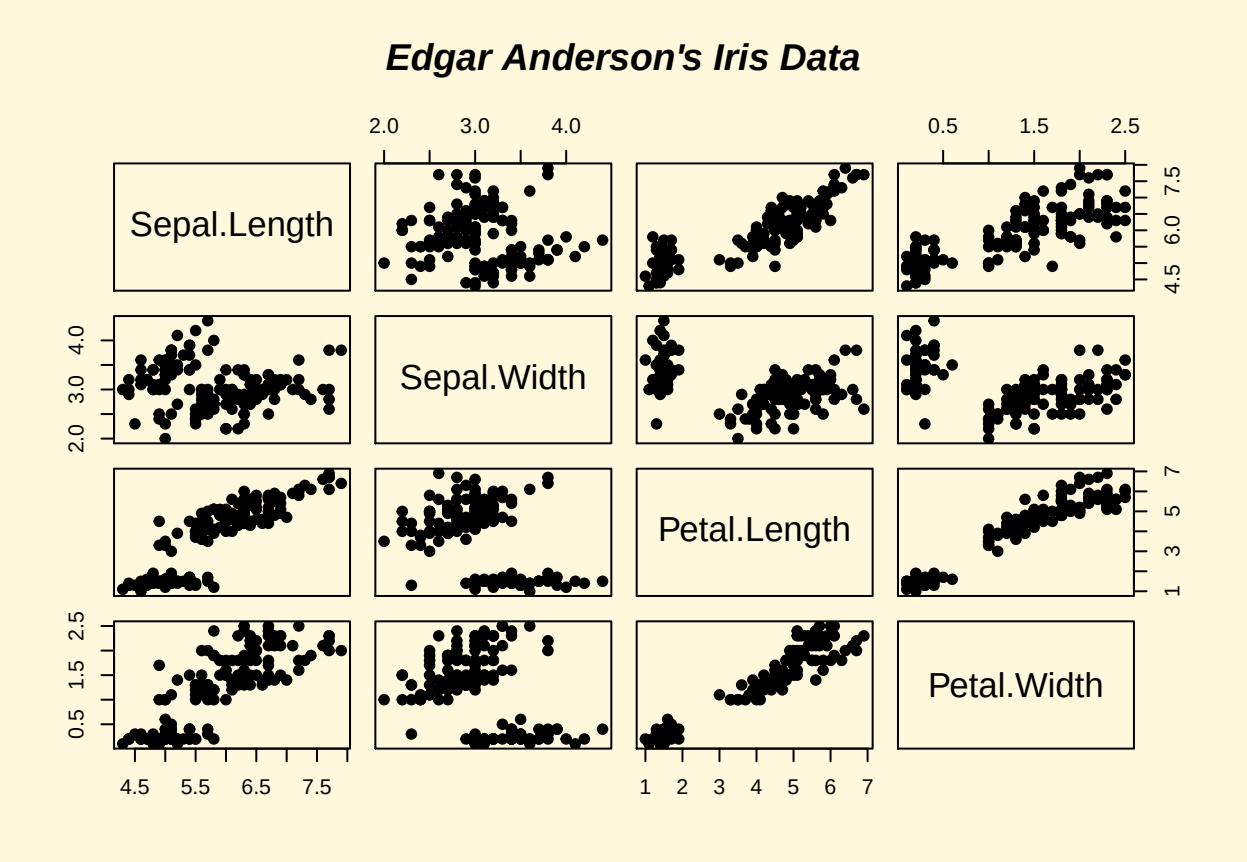



```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="cornsilk", border="black")
##
## > lines(x, col="blue")
##
## > points(x, pch=21, bg="lightcyan", cex=1.25)
##
## > axis(2, col.axis="blue", las=1)
##
## > axis(1, at=1:12, lab=month.abb, col.axis="blue")
##
## > box()
##
## > title(main= "The Level of Interest in R", font.main=4, col.main="red")
##
## > title(xlab= "1996", col.lab="red")
##
## > ## A filled histogram, showing how to change the font used for the
## > ## main title without changing the other annotation.
## >
## > par(bg="cornsilk")
##
## > x <- rnorm(1000)
##
```

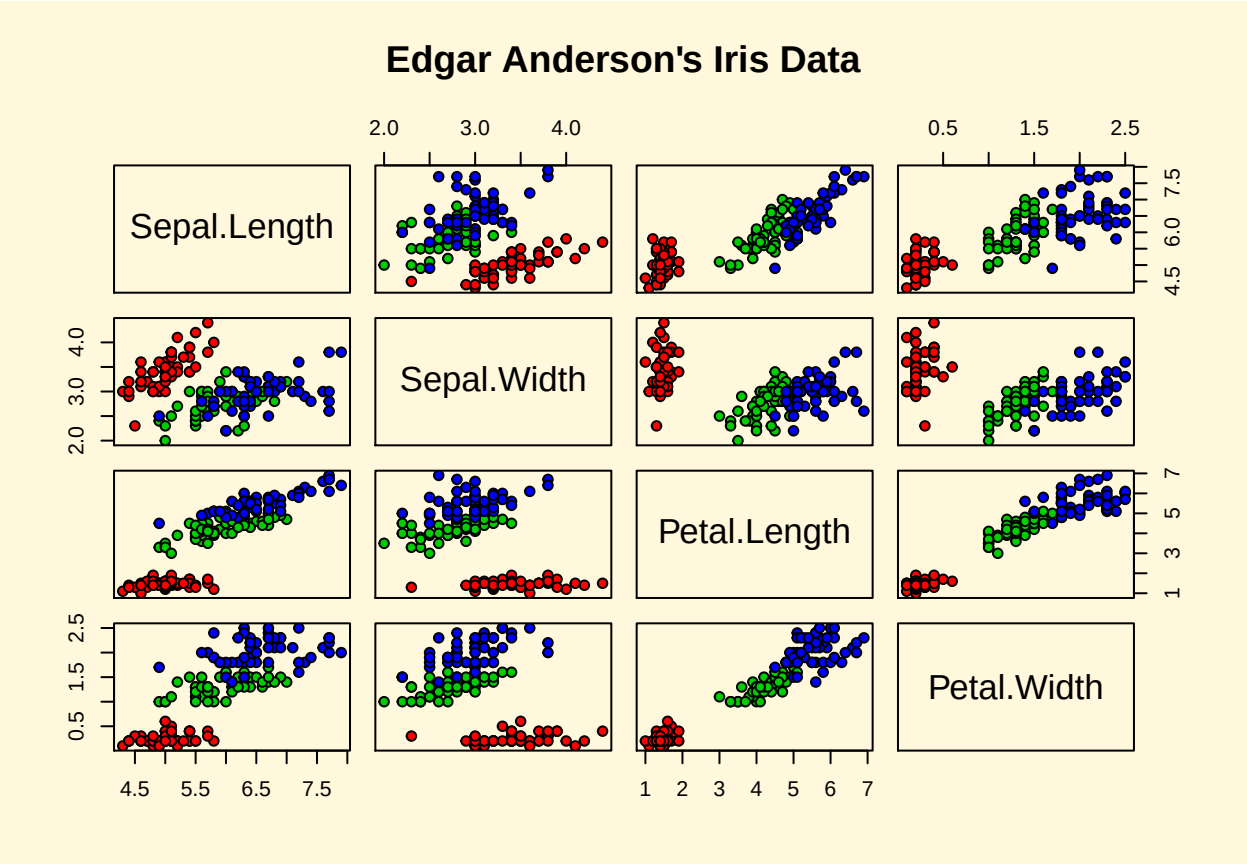
```
## > hist(x, xlim=range(-4, 4, x), col="lavender", main="")
```




```
##
## > title(main="1000 Normal Random Variates", font.main=3)
##
## > ## A scatterplot matrix
## > ## The good old Iris data (yet again)
## >
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", font.main=4, pch=19)
```

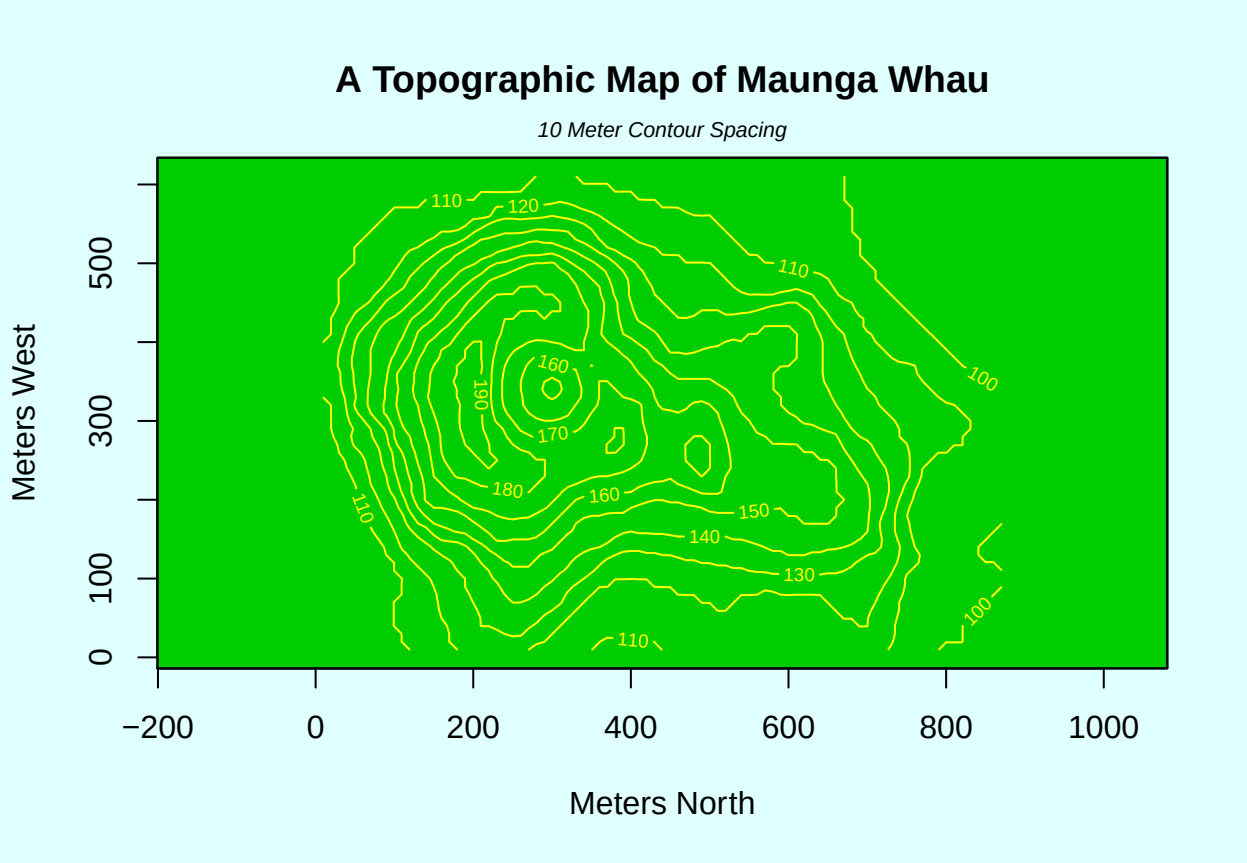



```
##
## > pairs(iris[1:4], main="Edgar Anderson's Iris Data", pch=21,
## +       bg = c("red", "green3", "blue")[unclass(iris$Species)])
```



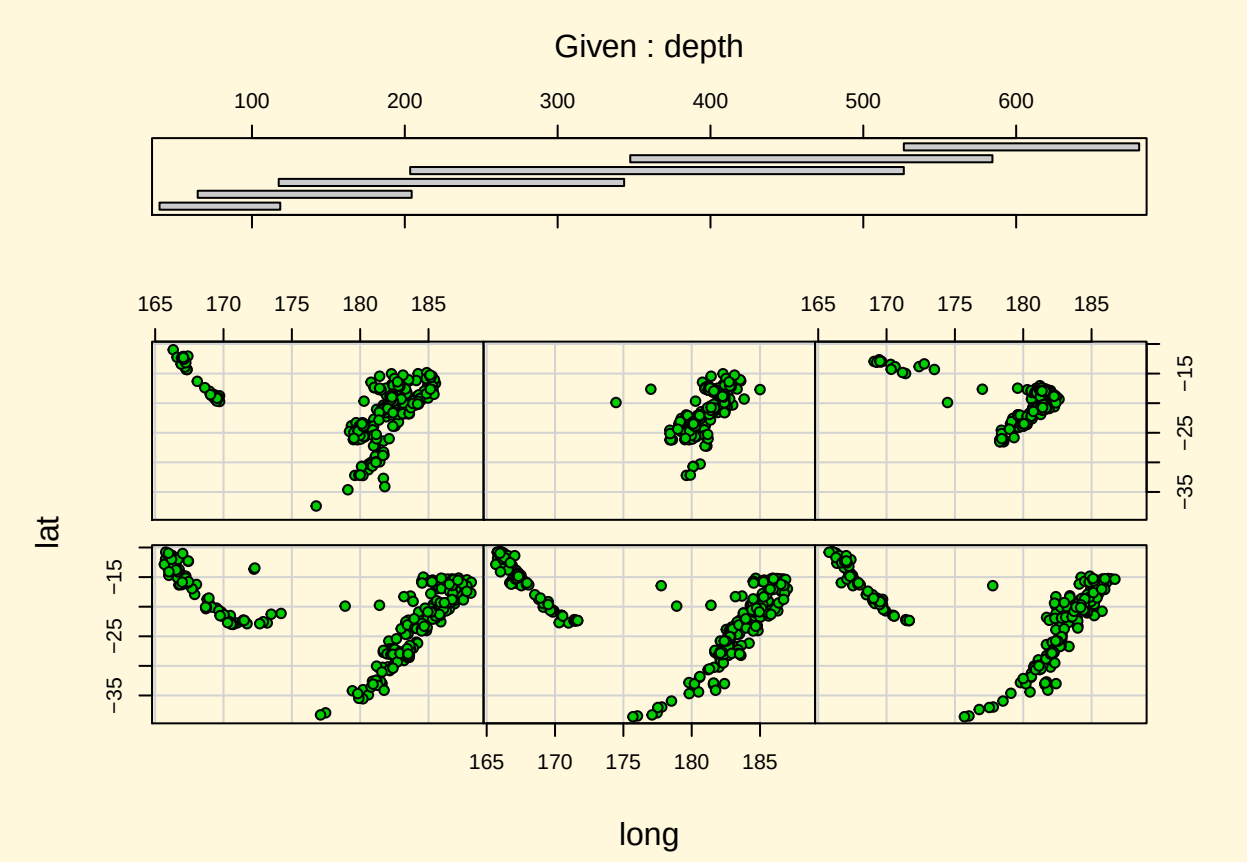

```
##
## > ## Contour plotting
## > ## This produces a topographic map of one of Auckland's many volcanic "peaks".
## >
## > x <- 10*1:nrow(volcano)
##
## > y <- 10*1:ncol(volcano)
##
## > lev <- pretty(range(volcano), 10)
##
## > par(bg = "lightcyan")
##
## > pin <- par("pin")
##
## > xdelta <- diff(range(x))
##
## > ydelta <- diff(range(y))
##
## > xscale <- pin[1]/xdelta
##
## > yscale <- pin[2]/ydelta
##
## > scale <- min(xscale, yscale)
##
## > xadd <- 0.5*(pin[1]/scale - xdelta)
##
```

```
## > yadd <- 0.5*(pin[2]/scale - ydelta)
##
## > plot(numeric(0), numeric(0),
## +       xlim = range(x)+c(-1,1)*xadd, ylim = range(y)+c(-1,1)*yadd,
## +       type = "n", ann = FALSE)
```




```
##
## > usr <- par("usr")
##
## > rect(usr[1], usr[3], usr[2], usr[4], col="green3")
##
## > contour(x, y, volcano, levels = lev, col="yellow", lty="solid", add=TRUE)
##
## > box()
##
## > title("A Topographic Map of Maunga Whau", font= 4)
##
## > title(xlab = "Meters North", ylab = "Meters West", font= 3)
##
## > mtext("10 Meter Contour Spacing", side=3, line=0.35, outer=FALSE,
## +       at = mean(par("usr")[1:2]), cex=0.7, font=3)
##
## > ## Conditioning plots
## >
## > par(bg="cornsilk")
##
```

```
## > coplot(lat ~ long | depth, data = quakes, pch = 21, bg = "green3")
```

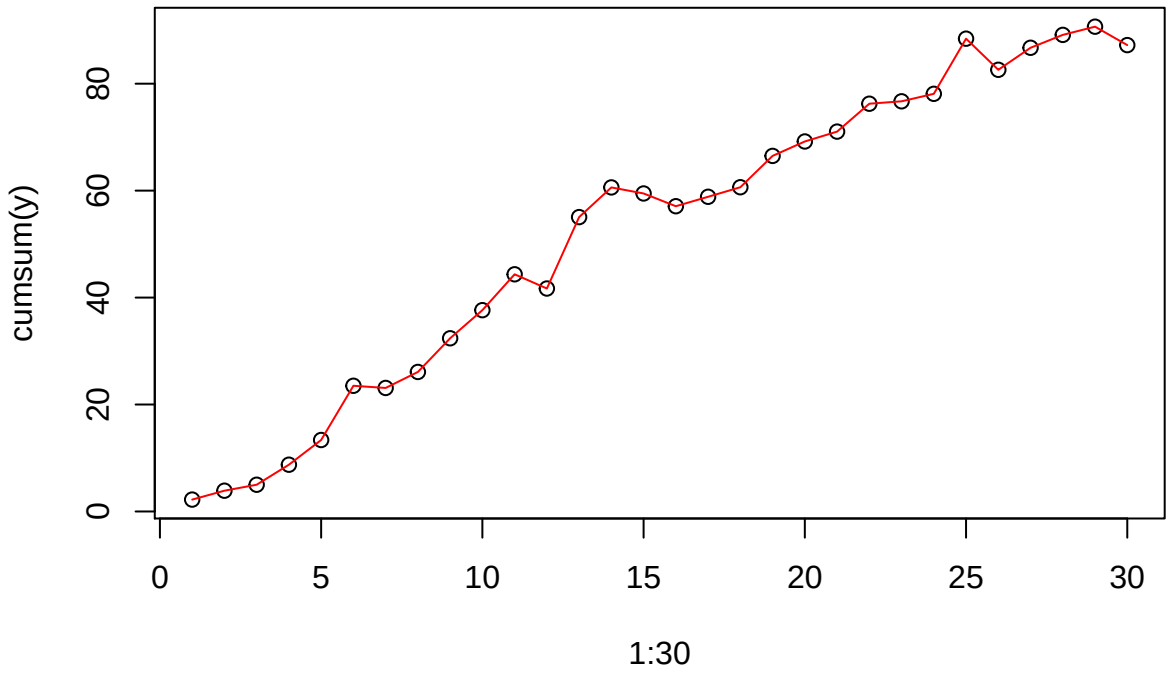



```
##
## > par(opar)
```

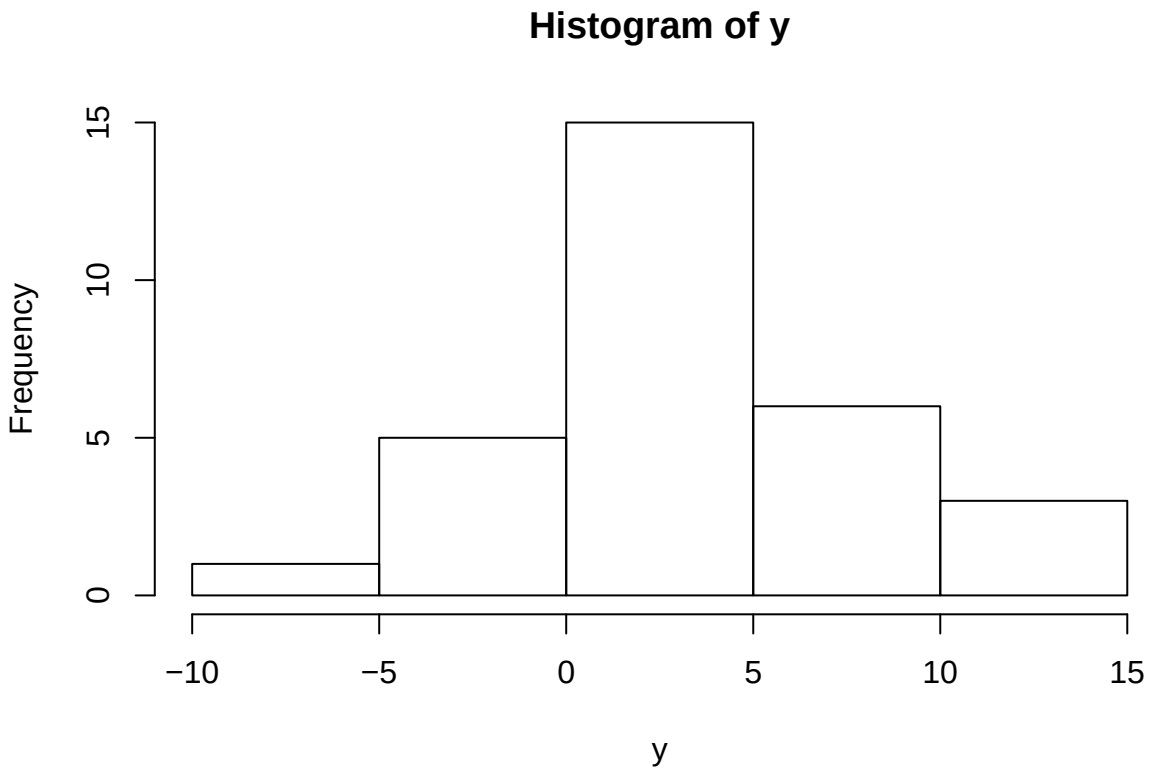
and keep pressing the return button on your keyboard until you scan through them all.

Let us use some of the basic functions for creating plots that will open a graph window in which the plot will be shown. Type the following and see what happens. Try to explain to yourself what the graph is showing.

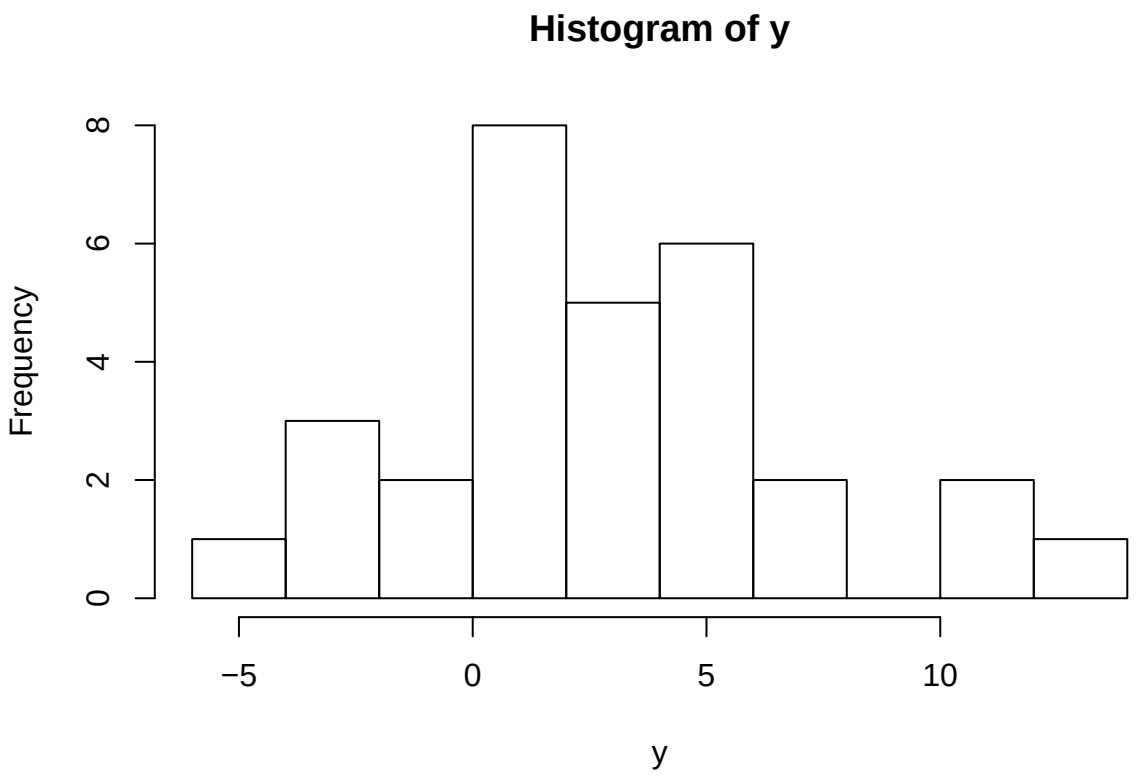
```
y <- rnorm(1:30, 2, 4)
plot(1:30, cumsum(y))
lines(1:30, cumsum(y), type="l", col="red")
```



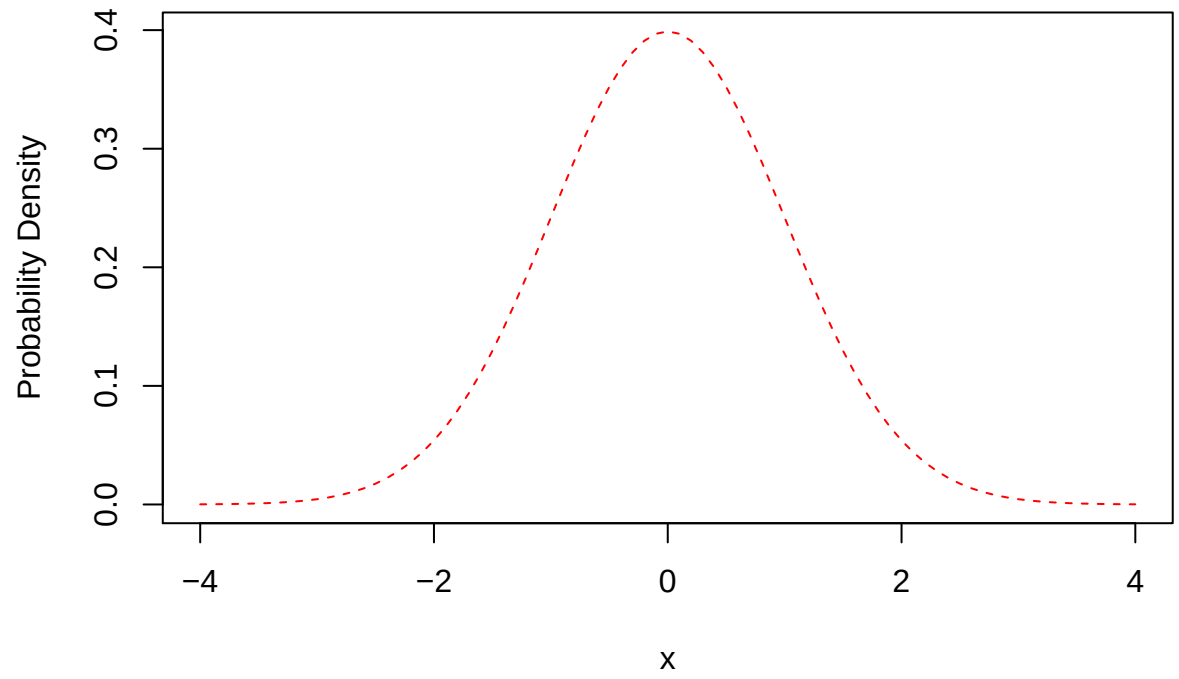

```
hist(y)
```



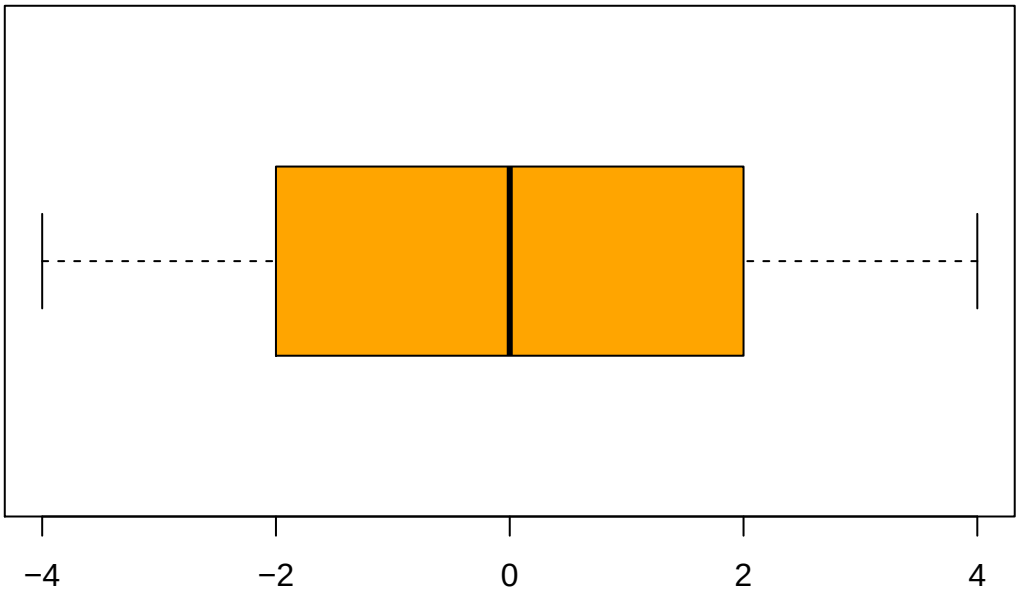

```
hist(y, 10)
```




```
x <- seq(-4, 4, 0.1)
plot(x, dnorm(x), type="l", lty=2, col="red", ylab="Probability Density", xlab="x")
```




```
boxplot(x, col="orange", horizontal=T)
```



Graphs are usually used to provide visualisation of specific features contained within the data, thus they can help us to communicate information. Often we can inspect the main characteristic of the data by using the appropriate plots. Questions that you should seek the answers to before embarking on producing any kind of plot are:

- What type of information one (or more) variables are providing: barchart or histogram?
- How two variables are related: boxplots or scatterplots?

In other words, to use the appropriate plot we need to understand the problem under investigation for which the data is collected and to recognise clearly what each of the variables in the data is measuring.

Your Turn

Before you start remove everything from your *workspace* in R by typing:

```
rm(list = ls())
```

Practise by doing the following set of exercises:

1) Carry out the following calculations:

- 4^5
- Add 7 to 9 and then divide the answer by 2
- Subtract π from 5 and raise answer to the power of 3 and then add 2.79
- Divide 6^8 by 4.7 and then divide the answer by 2

2) Create an object called "X1", which is 99.

3) Create an object called "X2", which is answer to 4^5.

4) Multiply X1 and X2 and calculate the square root of the answer which should be stored as a new object called "X3".

5) Calculate the *log* to the base of 10 of X1 and round it to the second decimal place.

6) Create vectors called "x1" and "x2", where vector x1 consists of numbers: 1, 4, 7, 9, 11, 12, 13, 15 and 18 and vector x2 of numbers: 1, 1, 1, 2, 2, 2, 3, 3, 3.

7) Subtract x2 from x1.

8) Create a new vector called "x3" by combining vectors x1 and x2.

9) Calculate mean and variance of x3.

10) Calculate medians for the three vectors.

11) Create a matrix called "m1" with the following elements:

$$\boldsymbol{m1} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}$$

12) Use a subscript to find the 2^{nd} number in vector x1 and x2 and element in the 2^{nd} raw and 3^{rd} column in matrix m1.

13) Add the 5^{th} number in vector x1 to the element in matrix m1 which is in 1^{st} raw and 1^{st} column.

14) Calculate the mean and the variance of all numbers in x3 that are less than 13.

15) Calculate the mean and the variance of all numbers in x3 that are greater than or equal to 3 and that are less than 12.