STATS SOLUTIONS

COUNT YOUR INTEREST BY STATISTICS.

Essentials For R Programming

ryannthegeek

2023-06-16

Contents

List of Figures

List of Tables

1 Introduction and Preliminaries

```
x <- 1:5
ls() ## lists objects formed
```

```
[1] "has_annotations" "x"
```

```
rm(x) ## remove objects formed
```

2 Simple Manipulations; Numbers and Vectors

To set up a vector named x consisting of five numbers, namely $10.4, 5.6, 3.1, 6.4$ and 21.7, use the R command

```
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Assignment can also be made using the function `assign()`. An equivalent way of making the same assignment as above is with:

```
assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

All of the common arithmetic functions are available:

```
log()
exp()
sin()
cos()
tan()
sqrt()
```

2.1 Logical vectors

The logical operators are:

```
<
<=
>
>=
== # equivalent to.
!=  # inequality.
```

In addition if `c1` and `c2` are logical expressions, then `c1 & c2` is their intersection and, on the other hand, `c1 | c2` is their union or, and `!c1` is the negation of `c1`.

2.2 Index vectors; selecting and modifying subsets of a data set

```
y <- x[!is.na(x)] # creates (or re-creates) an object y which will contain
   the non-missing values of x, in the same order

(x + 1)[(!is.na(x)) & x > 0] -> z # creates an object z and places in it the
    values of the vector x+1 for which the corresponding value in x was both
    non-missing and positive

x[is.na(x)] <- 0 # replaces any missing values in x by zeros and
y[y < 0] <- -y[y < 0] # has the same effect as
y <- abs(y)
```

3 Objects, their Modes and Attributes

3.1 Getting and setting attributes

The function `attributes(object)` returns a list of all the non-intrinsic attributes currently defined for that object. The function `attr(object, name)` can be used to select a specific attribute.

```
attr(z, "dim") <- c(10,10) # allows R to treat z as if it were a 10-by-10
   matrix
```

4 Ordered and Non-ordered Factors

4.1 A specific example

Suppose, for example, we have a sample of 30 tax accountants from all the states and territories of Australia1 and their individual state of origin is specified by a character vector of state mnemonics as:

```
state <- c("tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",
           "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",
           "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",
           "sa", "act", "nsw", "vic", "vic", "act")

## A factor is created using the factor() function:
statef <- factor(state)
```

4.2 The function tapply() and ragged arrays

Suppose we have the incomes of the same tax accountants in another vector (in suitably large units of money)

```
incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
             61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
             59, 46, 58, 43)
```

To calculate the sample mean income for each state we can now use the special function `tapply()`:

```
incmeans <- tapply(incomes, statef, mean);incmeans
```

```
     act      nsw       nt      qld       sa      tas      vic       wa
44.50000 57.33333 55.50000 53.60000 55.00000 60.50000 56.00000 52.25000
```

The function `tapply()` is used to apply a function, here `mean()`, to each group of components of the first argument, here incomes, defined by the levels of the second component, here $statef2$, as if they were separate vector structures.

Suppose further we needed to calculate the standard errors of the state income means. To do this we need to write an R function to calculate the **standard error** for any given vector.

```
stdError <- function(x) sqrt(var(x)/length(x))

## After this assignment, the standard errors are calculated by
incster <- tapply(incomes, statef, stdError);incster
```

```
     act      nsw       nt      qld       sa      tas      vic       wa
1.500000 4.310195 4.500000 4.106093 2.738613 0.500000 5.244044 2.657536
```

4.3 Frequency tables from factors

The function `table()` allows frequency tables to be calculated from equal length factors. If there are k factor arguments, the result is a $k - way$ array of frequencies.

```
statefr <- table(statef) # gives in statefr a table of frequencies of each
   state in the sample
```

The frequencies are ordered and labelled by the levels attribute of the factor. Further suppose that incomef is a factor giving a suitably defined income class for each entry in the data vector, for example with the `cut()` function:

```
 factor(cut(incomes, breaks = 35 + 10*(0:7))) -> incomef
# Then to calculate a two-way table of frequencies:
table(incomef, statef)
```

```
        statef
incomef   act nsw nt qld sa tas vic wa
  (35,45]   1   1  0   1  0   0   1  0
  (45,55]   1   1  1   1  2   0   1  3
  (55,65]   0   3  1   3  2   2   2  1
  (65,75]   0   1  0   0  0   0   1  0
```

5 Arrays and Matrices

5.1 Arrays

An array can be considered as a multiply sub scripted collection of data entries, for example numeric.

A dimension vector is a vector of non-negative integers. If its length is 4 then the array is $k - dimensional$, e.g. a matrix is a $2 - dimensional$ array. The dimensions are indexed from one up to the values given in the dimension vector.

5.2 Index matrices

Suppose for example;

we have a 4 by 5 array X and we wish to do the following:

1. Extract elements `X[1,3]`, `X[2,2]` and `X[3,1]` as a vector structure.
2. Replace these entries in the array X by zeroes.

In this case we need a 3 by 2 subscript array, as in the following example

```
x <- array(1:20, dim = c(4, 5));x # Generate a 4 by 5 array
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

```
i <- array(c(1:3,3:1), dim = c(3, 2));i # i is a 3 by 2 index array.
```

```
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
```

```
x[i] # Extract those elements
```

```
[1] 9 6 3
```

```
x[i] <- 0;x # Replace those elements by zeros.
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
```

```
##  The outer product of two arrays
```

If a and b are two numeric arrays, their outer product is an array whose dimension vector is obtained by concatenating their two dimension vectors (order is important), and whose data vector is got by forming all possible products of elements of the data vector of a with those of b.

The outer product is formed by the special operator `%o%`:

```
ab <- a %o% b
## An alternative is
ab <- outer(a, b, "*")
```

The multiplication function can be replaced by an arbitrary function of two variables. For example if we wished to evaluate the function $f(x;y) = cos(y)/(1 + x^2)$ over a regular grid of values with x and y coordinates defined by the R vectors x and y respectively, we could proceed as follows:

```
f <- function(x, y) cos(y)/(1 + x^2)
z <- outer(x, y, f)
```

5.3 Matrix facilities

As noted above, a matrix is just an array with two subscripts. However it is such an important special case it needs a separate discussion. R contains many operators and functions that are available only for matrices. For example `t(X)` is the matrix transpose function, as noted above.

The functions `nrow(A)` and `ncol(A)` give the number of rows and columns in the matrix A respectively.

5.3.1 Matrix multiplication

The operator `%*%` is used for matrix multiplication.

If, for example, A and B are square matrices of the same size, then

```
A <- matrix(data = 1:9, nrow = 3, ncol = 3)
B <- matrix(data = 2:10, nrow = 3, ncol = 3)
x <- 1:3
A * B # is the matrix of element by element products
```

```
     [,1] [,2] [,3]
[1,]    2   20   56
[2,]    6   30   72
[3,]   12   42   90
```

```
A %*% B # is the matrix product
```

```
     [,1] [,2] [,3]
[1,]   42   78  114
[2,]   51   96  141
[3,]   60  114  168
```

```
#  If x is a vector, then
x %*% A %*% x # s a quadratic form.
```

```
     [,1]
[1,]  228
```

The function `crossprod()` forms crossproducts, meaning that `crossprod(X, y)` is the same as `t(X) %*% y` but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

The meaning of `diag()` depends on its argument. `diag(v)`, where v is a vector, gives a diagonal matrix with elements of the vector as the diagonal entries. On the other hand `diag(M)`, where M is a matrix, gives the vector of main diagonal entries of M. if k is a single numeric value then `diag(k)` is the *k by k* identity matrix!.

5.3.2 Eigenvalues and Eigenvectors

The function `eigen(Sm)` calculates the eigenvalues and eigenvectors of a symmetric matrix Sm. The result of this function is a list of two components named *values* and *vectors*. The assignment:

```
A <- matrix(data = 1:9, nrow = 3, ncol = 3)
eigen(A)
```

```
eigen() decomposition
$values
[1]  1.611684e+01 -1.116844e+00 -5.700691e-16

$vectors
           [,1]       [,2]       [,3]
[1,] -0.4645473 -0.8829060  0.4082483
[2,] -0.5707955 -0.2395204 -0.8164966
[3,] -0.6770438  0.4038651  0.4082483
```

```
ev <- eigen(A)
ev$values #  is the vector of eigenvalues of Sm
```

```
[1]  1.611684e+01 -1.116844e+00 -5.700691e-16
```

```
ev$vectors #  is the matrix of corresponding eigenvectors
```

```
           [,1]       [,2]       [,3]
[1,] -0.4645473 -0.8829060  0.4082483
[2,] -0.5707955 -0.2395204 -0.8164966
[3,] -0.6770438  0.4038651  0.4082483
```

```
svd(A) #  Singular value decomposition and determinants
```

```
$d
[1] 1.684810e+01 1.068370e+00 5.039188e-17

$u
           [,1]        [,2]       [,3]
[1,] -0.4796712  0.77669099  0.4082483
[2,] -0.5723678  0.07568647 -0.8164966
[3,] -0.6650644 -0.62531805  0.4082483

$v
           [,1]       [,2]       [,3]
[1,] -0.2148372 -0.8872307 -0.4082483
[2,] -0.5205874 -0.2496440  0.8164966
[3,] -0.8263375  0.3879428 -0.4082483
```

6 Data and Data Manipulation

1. **matrix (base):** This is the basic matrix format and is based on the numeric index of rows and columns. This format is strict about the data class, and it isn't possible to combine multiple classes in the same table. For example, it is not possible to have both numeric and strings at the same table.
2. **data.frame (base):** This is one of the most popular tabular formats in R. This is a more progressive and liberal version of the matrix function. It includes additional attributes, which support the combination of multiple classes in the same table and different indexing methods.
3. **tibble (tibble):** It is part of the tidyverse family of packages (RStudio designed packages for data science applications). This type of data is another tabular format and an improved version of the data.frame base package with the improvements that are related to printing and sub-setting applications.
4. **ts (stats) and mts (stats):** This is R's built-in function for time series data, where ts is designed to be used with single time series data and multiple time series (mts) supports multiple time series data.
5. **zoo (zoo) and xts (xts):** Both are designated data structures for time series data and are based on the matrix format with a timestamp index.

6.1 Lists and data frames

6.1.1 list

An R list is an object consisting of an ordered collection of objects known as its components. There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. Here is a simple example

```
lst <- list(name = "Fred", wife = "Mary", no.children = 3,
            child.ages = c(4, 7, 9));lst
```

```
$name
[1] "Fred"

$wife
[1] "Mary"

$no.children
[1] 3

$child.ages
[1] 4 7 9
```

Components are always numbered and may always be referred to as such. Thus if lst is the name of a list with four components, these may be individually referred to as `lst[[1]]`, `lst[[2]]`, `lst[[3]]` and `lst[[4]]`. If, further, `lst[[4]]` is a vector sub-scripted array then `lst[[4]][1]` is its first entry. If lst is a list, then the function ‘length(lst)‘ gives the number of (top level) components it has.

6.2 Constructing and modifying lists

New lists may be formed from existing objects by the function `list()`. An assignment of the form

```
lst <- list(name_1 = object_1, ..., name_m = object_m)
```

which sets up a list lst of m components using $object_1, ..., object_m$ for the components and giving them names as specified by the argument names, (which can be freely chosen). If these names are omitted, the components are numbered only.

Lists, can be extended by specifying additional components. For example

```
lst[5] <- list(matrix = "Mat")
```

7 Grouping, Loops and Conditional Execution

7.1 Loops

What are loops

A loop is a programming instruction that repeats until a specific condition is reached.

The loop executes a code block again and again until no further action is required.

Each time the code block within the loop is executed is called an iteration.

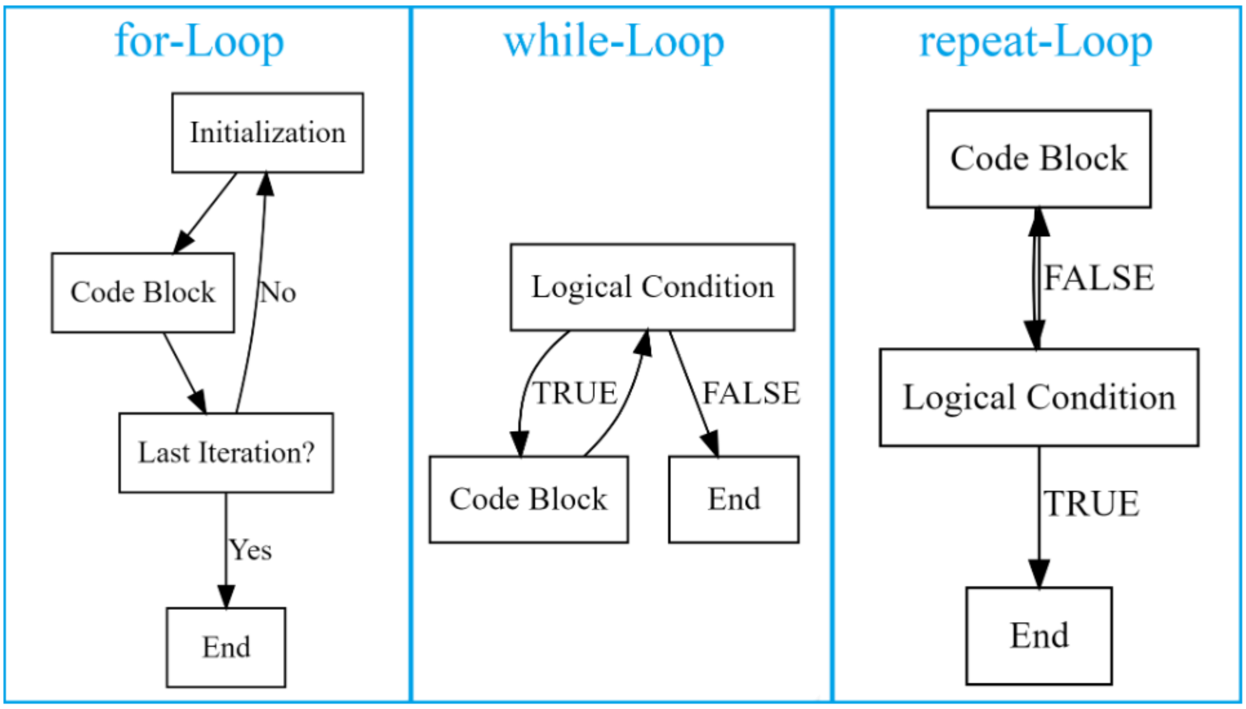

Figure 1: Loops in R

7.1.1 Writing for-Loops in R

for-loops specify a collection of objects (e.g. elements in a vector or list) to which a code block should be applied.

A for-loop consists of two parts: First, a header that specifies the collection of objects; Second, a body containing a code block that is executed once per object.

First, we have to specify a data object that we can use within the for-loop:

```
x_for <- 0    # Preliminary specification of data object
```

Let's assume that we want to run a for-loop that iterates over a vector with ten elements (i.e. 1:10). In each iteration, we want to add +1 to our data object and we want to print this data object

```
for (i in 1:10) {          # Head of for-loop
  x_for <- x_for + 1      # Body of for-loop
  print(x_for)
}
```

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

Click here to find more detailed explanations and advanced programming examples of for-loops in R.

7.1.2 Writing while-Loops in R

while-loops repeat a code block as long as a certain logical condition is TRUE.

This code is typically used when we don't know the exact number of times our R code needs to be executed.

The following code illustrates how to write and use while-loops in R. Again, we have to create a data object first:

```
x_while <- 0         # Preliminary specification of data object
```

Now, let's assume that we want to repeat a code block, which adds +1 to our data object, as long as our data object is smaller than 10. We also want to print this data object at the beginning of each iteration

```
while (x_while < 10) {          # Head of while-loop
  x_while <- x_while + 1       # Body of while-loop
  print(x_while)
}
```

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

Click here to find more detailed explanations and advanced programming examples of while-loops in R.

7.1.3 Writing repeat-Loops in R

repeat-loops repeat a code block until a break condition is fulfilled. This break condition marks the end of the loop.

repeat-loops follow a similar logic as while-loops, since they can also be used when the user doesn't know the exact number of times the R code should be repeated.

```
x_repeat <- 0                  # Preliminary specification of data object
```

Now, we can apply a repeat-loop to get the same output as in the previous examples as shown below:

```
repeat {                       # Head of repeat-loop
  x_repeat <- x_repeat + 1     # Body of repeat-loop
  print(x_repeat)
  if (x_repeat >= 10) {         # Break condition of repeat-loop
    break
  }
}
```

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

Click here to find more detailed explanations and advanced programming examples of repeat-loops in R.

8 Writing your own Functions

```
x <- 1:5
circle <- function(r) {
  Area <- pi*r^2
  Circumference <- 2*pi*r
  return(list(Area = Area, Circumference = Circumference))
}
circle(x)
```

```
$Area
[1]  3.141593 12.566371 28.274334 50.265482 78.539816

$Circumference
[1]  6.283185 12.566371 18.849556 25.132741 31.415927
```

```
x <- 1:5
circle <- function(r) {
  Area <- pi*r^2
  Circumference <- 2*pi*r
  return(data.frame(Area = Area, Circumference = Circumference))
}

output <- circle(x)
require(kableExtra)
kable(output)
```

Area	Circumference
3.141593	6.283185
12.566371	12.566371
28.274334	18.849556
50.265482	25.132741
78.539816	31.415927

9 Statistical Analysis, Tests and Distributions

9.1 Statistical Analysis

9.1.1 Central Moment

The k^{th} central moment of a data population can be obtained in R by:

Problem

Find the third central moment of eruption duration in the faithful dataset

```
require(e1071) # load e1071
duration = faithful$eruptions # eruption duration
moment(duration, order=3, center=TRUE)
```

```
[1] -0.6149059
```

9.2 Statistical Tests

9.2.1 Augmented Dickey–Fuller test

In statistics and econometrics, an augmented Dickey–Fuller test (ADF) tests the null hypothesis that a unit root is present in a time series sample.

9.3 Probability Distributions

9.3.1 R as a set of statistical tables

One convenient use of R is to provide a comprehensive set of statistical tables. Functions are provided to evaluate the cumulative distribution function $P(X \leq x)$, the probability density function and the quantile function (given q the smallest x such that $P(X \leq x) > q$), and to simulate from the distribution.

Table 1: Distributions in R

Distribution	R name	additional arguments
beta	`beta`	`shape1, shape2, ncp`
binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
chi-squared	`chisq`	`df , ncp`
exponential	`exp`	`rate`
F	`f`	`df1, df2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`prob`
hypergeometric	`hyper`	`m, n, k`
log-normal	`lnorm`	`meanlog, sdlog`
logistic	`logis`	`location, scale`
negative binomial	`nbinom`	`size, prob`
normal	`norm`	`mean, sd`

Distribution	R name	additional arguments
Poisson	`pois`	`lambda`
signed rank	`signrank`	`n`
Student's t	`t`	`df , ncp`
uniform	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

Prefix the name given here by `d` for the density, `p` for the CDF, `q` for the quantile function and `r` for simulation (random deviates). The first argument is `x` for `dxxx`, `q` for `pxxx`, `p` for `qxxx` and `n` for `rxxx` (except for `rhyper`, `rsignrank` and `rwilcox`, for which it is `nn`). In not quite all cases is the non-centrality parameter `ncp` currently available.
The `pxxx` and `qxxx` functions all have logical arguments `lower.tail` and `log.p` and the `dxxx` ones have `log`. This allows, e.g., getting the cumulative (or "integrated") hazard function, $H(t) = -log(1 - F(t))$, by

```
-pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
```

or more accurate log-likelihoods by

```
dxxx(..., log = TRUE)
```

In addition, there are functions `ptukey` and `qtukey` for the distribution of the studentized range of samples from a normal distribution, and `dmultinom` and `rmultinom` for the multinomial distribution. Further distributions are available in contributed packages, notably `Suppdists`. Here are some examples:

```
## 2-tailed p-value for t distribution
2 * pt(-2.43, df = 13)
```

```
[1] 0.0303309
```

```
## upper 1% point for an F(2, 7) distribution
qf(0.01, 2, 7, lower.tail = FALSE)
```

```
[1] 9.546578
```

9.4 Examining the distribution of a set of data

10 Appendix: All code for this report

```
x <- 1:5
ls() ## lists objects formed
rm(x) ## remove objects formed
x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
log()
exp()
sin()
cos()
tan()
sqrt()
```

```
<
<=
>
>=
== # equivalent to.
!=  # inequality.
y <- x[!is.na(x)] # creates (or re-creates) an object y which will contain
   the non-missing values of x, in the same order

(x + 1)[(!is.na(x)) & x > 0] -> z # creates an object z and places in it the
    values of the vector x+1 for which the corresponding value in x was both
    non-missing and positive

x[is.na(x)] <- 0 # replaces any missing values in x by zeros and
y[y < 0] <- -y[y < 0] # has the same effect as
y <- abs(y)
attr(z, "dim") <- c(10,10) # allows R to treat z as if it were a 10-by-10
   matrix
state <- c("tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",
           "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",
           "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",
           "sa", "act", "nsw", "vic", "vic", "act")

## A factor is created using the factor() function:
statef <- factor(state)
incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
             61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
             59, 46, 58, 43)
incmeans <- tapply(incomes, statef, mean);incmeans
stdError <- function(x) sqrt(var(x)/length(x))

## After this assignment, the standard errors are calculated by
incster <- tapply(incomes, statef, stdError);incster
statefr <- table(statef) # gives in statefr a table of frequencies of each
   state in the sample
 factor(cut(incomes, breaks = 35 + 10*(0:7))) -> incomef
# Then to calculate a two-way table of frequencies:
table(incomef, statef)
x <- array(1:20, dim = c(4, 5));x # Generate a 4 by 5 array

i <- array(c(1:3,3:1), dim = c(3, 2));i # i is a 3 by 2 index array.

x[i] # Extract those elements

x[i] <- 0;x # Replace those elements by zeros.

##  The outer product of two arrays
ab <- a %o% b
## An alternative is
ab <- outer(a, b, "*")
f <- function(x, y) cos(y)/(1 + x^2)
z <- outer(x, y, f)
A <- matrix(data = 1:9, nrow = 3, ncol = 3)
B <- matrix(data = 2:10, nrow = 3, ncol = 3)
x <- 1:3
A * B # is the matrix of element by element products
A %*% B # is the matrix product
```

```
#  If x is a vector, then
x %*% A %*% x # s a quadratic form.
A <- matrix(data = 1:9, nrow = 3, ncol = 3)
eigen(A)
ev <- eigen(A)
ev$values #  is the vector of eigenvalues of Sm
ev$vectors #  is the matrix of corresponding eigenvectors


svd(A) #  Singular value decomposition and determinants
lst <- list(name = "Fred", wife = "Mary", no.children = 3,
            child.ages = c(4, 7, 9));lst
lst <- list(name_1 = object_1, ..., name_m = object_m)
lst[5] <- list(matrix = "Mat")
x_for <- 0   # Preliminary specification of data object
for (i in 1:10) {         # Head of for-loop
  x_for <- x_for + 1      # Body of for-loop
  print(x_for)
}
x_while <- 0         # Preliminary specification of data object
while (x_while < 10) {         # Head of while-loop
  x_while <- x_while + 1      # Body of while-loop
  print(x_while)
}
x_repeat <- 0                # Preliminary specification of data object
repeat {                     # Head of repeat-loop
  x_repeat <- x_repeat + 1    # Body of repeat-loop
  print(x_repeat)
  if (x_repeat >= 10) {        # Break condition of repeat-loop
    break
  }
}
x <- 1:5
circle <- function(r) {
  Area <- pi*r^2
  Circumference <- 2*pi*r
  return(list(Area = Area, Circumference = Circumference))
}
circle(x)
x <- 1:5
circle <- function(r) {
  Area <- pi*r^2
  Circumference <- 2*pi*r
  return(data.frame(Area = Area, Circumference = Circumference))
}

output <- circle(x)
require(kableExtra)
kable(output)
require(e1071) # load e1071
duration = faithful$eruptions # eruption duration
moment(duration, order=3, center=TRUE)
-pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)
dxxx(..., log = TRUE)
## 2-tailed p-value for t distribution
2 * pt(-2.43, df = 13)
```

```r
## upper 1% point for an F(2, 7) distribution
qf(0.01, 2, 7, lower.tail = FALSE)
```