Introduction to the R Statistical Language

Last Updated: January 10, 2006

1 The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has:

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

2 Obtaining R

R is free and available to run on multiple platforms (Windows, Mac, Linux). R can be obtained by downloading and installing it from the R website at:

`http://www.r-project.org/`

After going to this page, click on the link `CRAN` on the left side of the page.

3 Getting help with functions and features

To get more information on any specific named function, for example `solve`, the command is

```
> help(solve)
```

An alternative is

```
> ?solve
```

For a feature specified by special characters, the argument must be enclosed in double or single quotes, making it a character string: This is also necessary for a few words with syntactic meaning including if, for and function.

```
> help("[[")
```

Either form of quote mark may be used to escape the other, as in the string ”It’s important”. Our convention is to use double quote marks for preference.

On most R installations help is available in HTML format by running

```
> help.start()
```

which will launch a Web browser that allows the help pages to be browsed with hyperlinks. On UNIX, subsequent help requests are sent to the HTML-based help system. The ‘Search Engine and Keywords’ link in the page loaded by help.start() is particularly useful as it is contains a high-level concept list which searches though available functions. It can be a great way to get your bearings quickly and to understand the breadth of what R has to offer.

The help.search command allows searching for help in various ways: try ?help.search for details and examples.

The examples on a help topic can normally be run by

```
> example(topic)
```

Windows versions of R have other optional help systems: use

```
> ?help
```

for further details.

4 Data Types and Manipulation

4.1 Vectors and assignment

R operates on named data structures. The simplest such structure is the numeric vector, which is a single entity consisting of an ordered collection of numbers. To set up a vector named x, say, 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)
```

Note Simply typing the name of an object will print the object. For example

```
> x
[1] 10.4  5.6  3.1  6.4 21.7
```

This is an assignment statement using the function `c()` which in this context can take an arbitrary number of vector arguments and whose value is a vector got by concatenating its arguments end to end.3

Note A number occurring by itself in an expression (a scalar) is taken as a vector of length one.

Notice that the assignment operator (<-), which consists of the two characters < (less than) and - (minus) occurring strictly side-by-side and it ‘points’ to the object receiving the value of the expression. In most contexts the = operator can be used as a alternative. 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))
```

The usual operator, <-, can be thought of as a syntactic short-cut to this.

Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

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

If an expression is used as a complete command, the value is printed and lost4. So now if we were to use the command

```
> 1/x
```

the reciprocals of the five values would be printed at the terminal (and the value of x, of course, unchanged).

The further assignment

```
> y <- c(x, 0, x)
```

would create a vector y with 11 entries consisting of two copies of x with a zero in the middle place.

4.2 Data types

There are three main data types that we will use in R: numeric, character, and logical. The three are relatively self-explanatory. Arithmetic procedures can be performed on numeric structures, character structures consist of text, and logical structures take on binary values (T vs. F).

```
#  Assign score to be a numeric vector
> score <- c(10.4, 5.6, 3.1, 6.4, 21.7)

#  Assign labs to be a numeric vector of labels
> labs <- c( "James", "Roger", "Brenda", "Marty", "Susan" )

#  Assign male to be a logical vector (Note this is case sensitive)
> male <- c( T, T, F, T, F)
```

Character vectors are often used for adding names to vectors

```
> names( score ) <- labs
> score
 James  Roger Brenda  Marty  Susan
  10.4    5.6    3.1    6.4   21.7
```

4.3 Sequences

R has a number of facilities for generating commonly used sequences of numbers. For example `1:30` is the vector `c(1, 2, ..., 29, 30)`.

Example 1 Put `n <- 10` and compare the sequences `1:n-1`, `1:(n-1)`, and `(n-1):1`.

The construction 30:1 may be used to generate a sequence backwards.

The function `seq()` is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is `seq(2,10)` is the same vector as `2:10`.

Parameters to `seq()`, and to many other R functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two parameters may be named from=value and to=value; thus `seq(1,30)`, `seq(from=1, to=30)` and `seq(to=30, from=1)` are all the same as `1:30`. The next two parameters to `seq()` may be named `by=value` and `length=value`, which specify a step size and a length for the sequence respectively. If neither of these is given, the default `by=1` is assumed. For example

```
> s3 <- seq(-5, 5, by=.2)
```

generates in s3 the vector c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0). Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

generates the same vector in s4.

The fifth parameter may be named along=vector, which if used must be the only parameter, and creates a sequence 1, 2, ..., length(vector), or the empty sequence if the vector is empty (as it can be).

A related function is rep() which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

which will put five copies of x end-to-end in s5. Another useful version is

```
> s6 <- rep(x, each=5)
```

which repeats each element of x five times before moving on to the next.

4.4 Vector arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element. Vectors occurring in the same expression need not all be of the same length. If they are not, the value of the expression is a vector with the same length as the longest vector which occurs in the expression. Shorter vectors in the expression are recycled as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated. So with the above assignments the command

```
> v <- 2*x + y + 1
```

generates a new vector v of length 11 constructed by adding together, element by element, `2*x` repeated 2.2 times, `y` repeated just once, and 1 repeated 11 times.

The elementary arithmetic operators are the usual `+, -, *, /` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log, exp, sin, cos, tan, sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of a vector respectively. range is a function whose value is a vector of length two, namely `c(min(x), max(x))`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x`, and `prod(x)` their product. Two statistical functions are `mean(x)` which calculates the sample mean, which is the same as `sum(x)/length(x)`, and `var(x)` which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance. If the argument to `var()` is an n-by-p matrix the value is a p-by-p sample covariance matrix got by regarding the rows as independent p-variate sample vectors.

Example 2 The pdf of a normall distributed random variable is given by the following expression:

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}$$

Use `rep` and `seq` to compute $f(x;\mu,\sigma)$ for 50 evenly spaced values between -3 and 3 for $\mu = 0$ and $\sigma = 1$. Store these values in a variable named `norm1`. Plot your values by typing the command:

```
> plot( x, norm1, type="l" )
```

4.5 Selection, logical values, and missing values

Elements of vectors may be indexed numerically (subscripting), or by their names, if names have been assigned.

```
#  Assign score to be a numeric vector
> score <- c(10.4, 5.6, 3.1, 6.4, 21.7)

#  Assign labs to be a numeric vector of labels
> labs <- c( "James", "Roger", "Brenda", "Marty", "Susan" )

#  Assign male to be a logical vector (Note this is case sensitive)
> male <- c( T, T, F, T, F)

#  Get the 2nd element of score
> score[2]
 Roger
 5.6

# Get everything except the 2nd element of score
> score[-2]
 James Brenda  Marty  Susan
  10.4    3.1    6.4   21.7

# Get elements 1, 3, and 4 from score
> score[ c(1,3,4) ]
 James Brenda  Marty
  10.4    3.1    6.4

# Get Marty and Roger's score
> score[ c( "Marty", "Roger") ]
 Marty Roger
   6.4   5.6
```

```
# Get all scores greater than 10
> score > 10
  James  Roger Brenda  Marty  Susan
   TRUE  FALSE  FALSE  FALSE   TRUE

> score[ score > 10 ]
 James Susan
  10.4  21.7

# Get all scores between 5 and 10
> score[ ( score > 5 ) & ( score < 10 ) ]
 Roger Marty
   5.6   6.4

# Get all scores less than 5 or greater than or equal to 10
> score[ ( score < 5 ) | ( score >= 10 ) ]
  James Brenda  Susan
   10.4    3.1   21.7
```

Missing values in R are denoted by NA. Here is an example of a missing value:

```
> score <- c( score, Warren=NA )

> score
  James  Roger Brenda  Marty  Susan Warren
   10.4    5.6    3.1    6.4   21.7     NA
```

Values can be selected based upon whether or not they are missing

```
# Get all missing scores
> score[ is.na(score) ]
  Warren
      NA

# Get all non-missing scores
> score[ !is.na(score) ]
  James  Roger Brenda  Marty  Susan
   10.4    5.6    3.1    6.4   21.7
```

4.6 Matrices and data frames

A matrix is a two dimensional array. A matrix must be entirely of the same data type: character, numeric, or logical. A matrix can be formed by 'binding' multiple vectors. For example

```
> score1 <- score

> score2 <- c( 12, 32.4, 13.1, 14, 17, 11.1 )

> score.matrix <- cbind( score1, score2 )

> score.matrix
            score1 score2
      James    10.4   12.0
      Roger     5.6   32.4
      Brenda    3.1   13.1
      Marty     6.4   14.0
      Susan    21.7   17.0
      Warren     NA   11.1
```

```
> score.matrix2 <- rbind( score1, score2 )

> score.matrix2
             James Roger Brenda Marty Susan Warren
      score1  10.4   5.6    3.1   6.4  21.7     NA
      score2  12.0  32.4   13.1  14.0  17.0   11.1
```

Elements of a matrix can be accessed by referring to the row and column which they reside in:

```
> score.matrix[1,2]
[1] 12

> score.matrix[,1]
 James  Roger Brenda  Marty  Susan Warren
  10.4    5.6    3.1    6.4   21.7     NA

> score.matrix[,2]
 James  Roger Brenda  Marty  Susan Warren
  12.0   32.4   13.1   14.0   17.0   11.1

> score.matrix[ 1:3, ]
       score1 score2
James    10.4   12.0
Roger     5.6   32.4
Brenda    3.1   13.1
```

A data frame is like a matrix, but it allow for different data types to be combined. As data are often of mixed types, data frames are very useful.

```
> male <- c( male, T )

> scores.dframe <- as.data.frame( cbind( male, score1, score2 ) )

> scores.dframe
       male score1 score2
James     1   10.4   12.0
Roger     1    5.6   32.4
Brenda    0    3.1   13.1
Marty     1    6.4   14.0
Susan     0   21.7   17.0
Warren    1     NA   11.1
```

Example 3 Using `sort` and `order`.

Included in R is a dataset call `trees`, which contains `Girth, Height`, and `Volume` measurements on 31 randomly sampled trees. To view the data, type `trees`.

Use the help function to learn about `sort` and `order`, rearrange the rows of `trees` so that they appear in a sorted order sorting first on `Height`, then on `Volume`, and finally on `Girth`. Store the sorted data in `trees.sort` and print out the first 10 rows of `trees.sort`.

5 Writing user defined functions

If you will be performing a series of operations multiple times, it is often a good idea to define a function that will perform each of the operations for you upon being called. Just as in mathematics, a function is composed of inputs and results in an output.

For a simple function, consider calculating the mean of a vector of numbers. The R predefined function `mean()` will do this for you:

```
> temp <- rnorm( 5, mean=1, sd=1 )
> temp
[1]  2.2707461 -0.5475682 -0.0554734 -2.0490441 -0.6653020
> mean( temp )
[1] -0.2093283
```

We can write our own mean function by relying upon the R function `sum()`:

```
mymean <- function( x ){
     rslt <- sum( x ) / length(x)
     return( rslt )
     }

> mymean( temp )
[1] -0.2093283
```

Notice that within our function will can call other pre-existing functions. A more complicated function might produce many statistics for a particular variable. Consider the `summary()` function in R:

```
> summary( temp )
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.04900 -0.66530 -0.54760 -0.20930 -0.05547  2.27100
```

Suppose that we would like to produce a more indepth description of our variable of interest (eg. producing the number missing, the mean, standard deviation, sample mode(s), inter-quartile range, etc). We can write a generic function to output all of these values via a single command. First, we will create a "helper function" that will calculate the modes of a generic variable `x`:

```
##### Definition of the function modes(x). Note that the
##### variable "x" is just a dummy variable used locally
##### in the definition of the function. This function will
##### return the mode(s) of the nonmissing values in a
##### vector that is supplied to it in place of "x".
#####
##### In this function, I define the modes as missing (NA)
##### if there is only one replicate of each nonmissing
##### value in x or if all the values of x are missing.

modes <- function (x){
  x <- x[!is.na(x)]                # removes missing values
  if (length(x) == 0)
    rslt <- NA                     # returns NA if no nonmissing values
  } else {
```

```
    tbl <- table (x)                # counts replicates for each value
    if (max (tbl) == 1) {
      rslt <- NA                    # no mode (1 replicate of each value in x)
     else {
      val <- as.numeric(names(tbl)) # unique values of x taken from the
                                    #  (character) names of tbl
      rslt <- val[tbl == max (tbl)] # return values of x which have the
                                    #   a number of replicates equal to
                                    #   the maximum in tbl
    }
  }
  rslt                              # the value of the last statement of
                                    #   a function is the returned value
}

##### The curly brackets "{" and "}" group statements into a
##### compound statement. If only one statement is desired, the
##### curly brackets are not truly necessary.
#####
##### In the above definition, I could have made the definition a bit
##### more concise, but I put in some unnecessary lines to improve human
##### readability. The following function does the exact same thing.
##### Note that this is officially a two-statement function in which
##### the second statement is a compound if-else statement. Thus the
##### function returns the value of that compound statement, which
##### is either the result NA if the length of x is 0, or it returns
##### the result of the else clause, which is a compound statement.
##### The result of that compound else clause is the result of the
##### last statement in the compound clause, which is the result
##### of the if-else: either NA or the vector of modes.

modes2 <- function (x)  {
  x <- x[!is.na(x)]
  if (length(x) == 0)  {
    NA                                  # no nonmissing values
   } else  {
    tbl <- table (x)                    # counts replicates
    if (max (tbl) == 1)  {
      NA                                # no mode
     } else  {
      as.numeric(names(tbl))[tbl==max(tbl)]  # modes
     }
   }
 }
```

Now we can write the function `describe()` which will generate our descriptive statistics:

```
##### Definition of the function describe(x).
##### This function will return
##### descriptive statistics for a numeric vector x (it assumes
##### that x is in fact numeric-- if it isn't, an error
##### will result).
#####
##### The descriptive statistics returned will be (in order):
##### - the length of x
##### - the number of missing values in x
##### - the mean of the nonmissing values in x
##### - the sample standard deviation of the nonmissing values in x
##### - the minimum nonmissing value in x
##### - the 25th percentile of the nonmissing values in x
##### - the median of the nonmissing values in x
##### - the 75th percentile of the nonmissing values in x
##### - the maximum nonmissing value in x
##### - the mode(s) of x
#####
```

```
##### An NA is returned for any of the above that are undefined.
#####
##### I make use of S-plus functions is.na(), mean(), var(), and
##### quantiles() (the latter of which by default returns the
##### min, max, and quartiles). I also call the function modes()
##### that I defined above.
#####
##### Note the way that I name the elements of the returned descriptive
##### statistics.

describe <- function (x)  {
  n <- length (x)                                # number of elements
  x <- x[!is.na(x)]                              # remove missing values
  nmsng <- n - length(x)                         # number missing
  if (length(x) == 0)  {
    rslt <- rep (NA, 10)                         # all NA's if no nonmissing
   } else  {
    xmode <- modes(x)                            # modes
    xquartiles <- quantile(x)                    # min, 25%ile, median,
                                                 #   75%ile, max
    xrange <- xquartiles[5] - xquartiles[1]      # range
    xIQrng <- xquartiles[4] - xquartiles[2]      # interquartile range
    rslt <- c(mean(x),                           # result combines mean,
          sqrt (var(x)),                         #    sample std dev,
          xquartiles,                            #    min, 25%ile, mdn,
                                                 #    75%ile, max,
          xIQrng,                                #    interquartile range,
          xrange,                                #    range,
          xmode)                                 #    modes
   }
  rslt <- c(n,                                   # combine sample size,
        nmsng,                                   #    number missing,
        rslt)                                    #    other descriptives
  names(rslt) <- c("N","Msng","Mean","Std Dev",  # define names for rslt
          "Min","25%ile","Median",
          "75%ile","Max","IQ rng","Range",
          rep("Mode",length(rslt)-11))           # may be multiple modes
  rslt
 }
```

Now we can try it out:

```
> describe( temp )
         N       Msng       Mean    Std Dev        Min     25%ile
 5.0000000  0.0000000 -0.2093283  1.5717278 -2.0490441  -0.6653020

     Median     75%ile        Max     IQ rng      Range       Mode
 -0.5475682 -0.0554734  2.2707461  0.6098286  4.3197902         NA
```