Invoking R: Click on the icon from your windows desktop

Exiting R: after you finish, type q()

The R prompt and the assignment operator

The R prompt looks like

> 

The assignment operator takes two strokes to create a pointer

>x <- 1 assigns the value 1 to x

Comments

Comment lines in R are denoted by #

# This is a comment line

For On-Line Help

Type, either

>help(command)

or

>?command

for example

>help(mean)

gives you information about the command mean, which takes the mean of a vector

Classes of objects in R

Vectors

>x <- c(1.1,2.1,4.5,6.1,8.9,10,1.2,
+ 4.3,6.1)

creates a numerical vector of length 9 containing the indicated values. Notice that if you run out of room on the screen entering your data, R automatically continues on the next line and inserts a “+” for you.

>y <- c(1:100)

creates a numerical vector of length 100, that contains the numbers 1 through 100.
> y <- vector("numeric", length=100)*0
creates a numerical vector of length 100 containing all zeros

> labels1 <- c("population", "illiteracy", "num. murders")
creates a vector of length 3 which is of mode character.

**Comparison operators**

(Using the previously defined vector x)

> x > 6

[1] FALSE FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE TRUE

The comparison operators are "<", ">", "<=", ">=", and equality and inequality tests are "=" and "!=".

**Using subscripts to extract data**

> x[2:4]

[1] 2.1 4.5 6.1

> x[x>8]

[1] 8.9 10

> x[-1]

[1] 2.1 4.5 6.1 8.9 10 4.3 6.1

The effect of the negative subscript is to give all but the first element.

**Operations on vectors**

The usual operators exist in R. For example,

element-by-element operators

> z <- x*y    # performs element by element multiplication
> z <- x^(1/2)    # raises each element to the \(\frac{1}{2}\) power (square root)
> z <- y+x    # does element by element addition
> z <- x/y    # performs element by element division

matrix operations

> t(x)    # takes the transpose of x
> t(x) %*% x  # multiplies the transpose of x by x
> x %*% t(x)  # multiplies the vector x by its transpose

Some useful vector commands

> length(x)  # gives the length of the vector
> paste(x,y)  # pastes the two vectors together

if x had the value “hello” and y had the value “world” then the paste command would yield “hello world”

e.g.

z <- paste(x,y)

> min(x)  # gives the minimum value of x
> max(x)  # gives the maximum value of x
> mean(x)  # gives the mean value of x
> median(x)  # gives the median value of x
> range(y)  # gives the range of y
> sqrt(z)  # performs element by element square root operation
> sum(x)  # sums the elements of x

Commands can be combined into expressions. For example,

> answer <- sqrt(sum(z-mean(z))^2)/(length(z)-1)

Some commands that return logical results are

> all(x>=0)
> any(x>=0)

To construct sample quantiles for a set of data

> quantile(xdata,c(0.25,0.75))

returns a vector containing the 25% and 75% quantiles in the data.

Sample variances are found by:

> var(x)  # gives the variance of the elements contained in x
> cor(x,y)  # gives the correlation between x and y

Consult the manual for many other useful commands.
Operations on Matrices

Matrices have two attributes: a dimension and (optionally) dimnames. The following are some examples of how to define matrices:

```r
> m <- matrix(0,3,4)  # gives a 3 x 4 matrix filled with zeros and stores it in m
> m <- matrix(1:12,3,4)
```

creates the matrix

```
 1 4 7 10 
 2 5 8 11 
 3 6 9 12
```

```r
matrix(1:12,ncol=3,byrow=T,dimnames=list(letters[1:4],LETTERS[1:3]))
```

creates the matrix

```
   A  B  C
a 1  2  3
b 4  5  6
c 7  8  9
d 10 11 12
```

Notice the option `byrow=T`, which tells the function to input the objects one row at a time instead of the default which is to do it one column at a time. This matrix has dimension names, which will be explained below.

There are functions available for testing whether an object is a matrix and for coercing an object to be interpreted as a matrix. For example,

```r
> is.matrix(x)  # returns a value TRUE of FALSE to test whether object is a matrix
> as.matrix(x)  # coerce an object x to be a matrix
```

```r
> dim(m) <- c(6,2)  # make m a 6x2 matrix
> dimnames(m) <- list(LETTERS[1:6],letters[1:2])  # assign names to the rows and columns
```

Subscripting

Suppose x is a matrix

```r
> y <- x[,c(1,3)]  # extracts first and third columns of x and puts them in y
# Notice that leaving the row field empty gives all rows
```

```r
> poor <- income.data[earnings <12000,]  # poor gets all rows of income data with earnings < $12,000
```
Functions on Matrices

> ncol(x)   # number of columns
> nrow(x)   # number of rows
> t(x)      # transpose
> cbind(…)  # combine columns, useful for creating matrices
    # cbind(x,y) – combines two vectors x and y to create a matrix that has
    # columns x and y
> rbind(…)  # combine rows
> diag(x)   # extract the diagonal elements of a matrix or create a diagonal matrix
> apply(x, m, fun)  # apply a function to the rows or columns of x
> sweep(a, m, stats)  # sweep out a row of row or column statistics

ex.

apply(z,2,mean)

takes the mean of z within each column

Some examples using these functions

> xnew <- cbind(1,x)  # adds a column of ones to the matrix x, stores result in xnew

To take the mean across columns

> col.means <- apply(x,2,mean)  # if the second field in the apply() function is 2 then the function is
    # applied across columns. If it is 1 then the function is applied across
    # rows.

> col.resid <- sweep(x,2,col.means)

This expands the col.means vector into a matrix and gives the difference between the original columns of x
and their means. The subtraction operator is the default, but other operators can be specified.

Some other useful matrix functions

> x %*% y  # matrix multiplication
> lsfit(x,y)  # least squares regression on y on x
    # Note that there is an alternative way of doing this using model
    # formulas
> solve(x,y)
> eigen(x)  # eigenvalues of x

There are many other matrix functions not listed here.

Arrays

A matrix is a two-way array. R can handle arrays of more than two dimensions. To create an array of size
50×4×3, filled with the contents of the vector school.data
> school.quality <- array(school.data,c(50,4,3))

To give names to the columns and rows, use a list of length k. For example


**Categorical Variables**

Categorical variables are ones like “male” and “female.” R has some useful features for generating categorical variables from continuous ones. For example,

> income.cat <- cut(income,4,c(“poor”, “lower.middle”, “upper.middle”, “upper”))

creates a categorical variable with four levels with the specified names. The range of income is cut into 4 equal length intervals, but the boundaries of the intervals could also have been specified.

> table(occupation,income.cat)

produces a two-way table of counts

<table>
<thead>
<tr>
<th></th>
<th>poor</th>
<th>lower.middle</th>
<th>upper.middle</th>
<th>upper</th>
</tr>
</thead>
<tbody>
<tr>
<td>students</td>
<td>99</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>financial.analysts</td>
<td>0</td>
<td>0</td>
<td>75</td>
<td>25</td>
</tr>
<tr>
<td>accountants</td>
<td>0</td>
<td>15</td>
<td>75</td>
<td>10</td>
</tr>
</tbody>
</table>

**Random Numbers**

R has a large number of built-in probability distributions and the capacity to generate random numbers from different distributions. The distributions include: beta, Cauchy, chi-square, exponential, F, Gamma, log-normal, logistic, normal, Student’s t, and uniform.

For example

> x <- rnorm(100,0,1)

takes 100 draws from a normal distribution with mean 0 and variance 1 and stores them in the vector x.

**Model formulas**

Fitting mathematical models, such as a least squares fit, requires the use of model formulae. In R, the operator “~” is used to designat雷 a model. For example,

`lnearn ~ education + experience + race + marital`

refers to a statistical model of the form

`lnearn = a + b1*education + b2*experience + b3*race + b4*marital + e`

Notice that there is no need to specify the parameters or the error term and that the intercept is in the model by default. You can define a model without an intercept by
\textbf{Inearn} \sim -1 + \textbf{education} + \textbf{experience} + \textbf{race} + \textbf{marital}

A variable in a model formulae may be a “factor” rather than numeric. A factor is an object that represents values from some specified set of possible levels. For example, suppose we have a variable sex which takes on the values “male” or “female”

\texttt{> Salary \sim Age + Sex}

refers to a model of the form

\begin{align*}
\text{Salary} &= b_1 + b_2 \ast \text{age} + b_3 \ \text{if “male”} + e \\
&\quad + b_4 \ \text{if “female”}
\end{align*}

\texttt{> Salary \sim Sex + poly(Age,3)}

would include a third-order polynomial in age in the regression.

\textit{Calling Model Fitting Functions}

All model fitting functions take a model formula as their first argument. For example, to do a least squares fit

\texttt{> lm(lnearn \sim education + experience + race + marital)}

\texttt{> mod1.res \leftarrow lm(lnearn \sim education + experience + race + marital)}

The second command stores the results from the model fit into mod1.res. To see the results, either type mod1.res or use the command

\texttt{> summary(mod1.res)}

which gives a more complete report of the least squares fit including the R-square value and the t-statistics.

\textit{More Complicated Formulas}

Interaction terms: the “:\”, “*” and “-” operators

\texttt{> lnearn \sim education + race + race:education}

will interact the different levels of education with those of experience to get the main effects and the interaction effects. A short-hand way of specifying the same model is

\texttt{> lnearn \sim education*race}

which again gives both the main effects and the interaction effects.

\texttt{> lnearn \sim education*race*experience}

would give all the main effects and all the interaction effects, including the third-order interaction term.

\texttt{> lnearn \sim education*race*experience-education:experience:race}

removes the third-order term and is equivalent to
> `lmearn ~ education + experience + race + education:race + education:experience + experience:race`

Updating Models: the "." operator

The "." operator is useful for updating a model by adding or subtracting regressors.

> `update(model1, . ~ . - Age)`

updates the linear model object by modifying its formula and then refitting it. Here, age is removed from the regression.

> `lm(Mileage ~ ., data = car.test.frame)`

fits the model using the data in car.test.frame. All the variables in car.test.frame are used as regressors except for Mileage which was specified as the regressand.

More on the data frame structure below.

Dealing with Missing Values when Fitting a Model

In R, missing values are designated by “NA”. To omit any observations which have an NA for one of the variables in the model fit, use the “na.action” option.

> `second.model <- lm(growth ~ lpwc1, data = lawdat, na.action=na.omit)`

Organizing Data for Models: Data Frames

A very useful structure for datasets in R is the data frame. A data frame is an object that represents a sequence of observations on some chosen variable. A data frame is similar to a matrix in that the variables can be treated as columns of the matrix and the observations as rows. It is different from a matrix in that the variables can also be treated as separate objects and the elements of the data frame need not all be of the same model. For example, one column could be character, another numeric, another logical.

Reading Tables of Data and Creating Data Frames

Suppose you have the following tables stored in an ascii or txt file “auto1” that you wish to read into a data frame

<table>
<thead>
<tr>
<th></th>
<th>Price</th>
<th>Country</th>
<th>Mileage</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acura.Integra</td>
<td>11950</td>
<td>Japan</td>
<td>NA</td>
<td>Small</td>
</tr>
<tr>
<td>Audi.100.5</td>
<td>26900</td>
<td>Germany</td>
<td>NA</td>
<td>Medium</td>
</tr>
<tr>
<td>BMW.325i.6</td>
<td>24650</td>
<td>Germany</td>
<td>NA</td>
<td>Compact</td>
</tr>
<tr>
<td>Chevrolet.Lumina.4</td>
<td>12140</td>
<td>USA</td>
<td>NA</td>
<td>Medium</td>
</tr>
<tr>
<td>Ford.Festiva.4</td>
<td>6319</td>
<td>Korea</td>
<td>37</td>
<td>Small</td>
</tr>
<tr>
<td>Mazda.929.V6</td>
<td>23300</td>
<td>Japan</td>
<td>21</td>
<td>Medium</td>
</tr>
<tr>
<td>Mazda.MX.5.Miata</td>
<td>13800</td>
<td>Japan</td>
<td>NA</td>
<td>Sporty</td>
</tr>
<tr>
<td>Nissan.300.ZX.V6</td>
<td>27900</td>
<td>Japan</td>
<td>NA</td>
<td>Sporty</td>
</tr>
<tr>
<td>Oldsmobile.Calais.4</td>
<td>9995</td>
<td>USA</td>
<td>23</td>
<td>Compact</td>
</tr>
<tr>
<td>Toyota.Cressida.6</td>
<td>21498</td>
<td>Japan</td>
<td>23</td>
<td>Medium</td>
</tr>
</tbody>
</table>

To read this into a data frame, use the read.table function
> somedata <- read.table("auto1")

The car names will be assigned as row names and the column names will be “Price”, “Country of Origin” “Mileage” and “Type.” To view the row and column names type

> dimnames(somedata)

You do not need to specify the column names and row names separately in the input file. They can also be assigned separately. If we had the file “auto2” containing

<table>
<thead>
<tr>
<th>Price</th>
<th>Country</th>
<th>Mileage</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11950</td>
<td>Japan</td>
<td>NA</td>
<td>Small</td>
</tr>
<tr>
<td>26900</td>
<td>Germany</td>
<td>NA</td>
<td>Medium</td>
</tr>
<tr>
<td>24650</td>
<td>Germany</td>
<td>NA</td>
<td>Compact</td>
</tr>
<tr>
<td>12140</td>
<td>USA</td>
<td>NA</td>
<td>Medium</td>
</tr>
<tr>
<td>6319</td>
<td>Korea</td>
<td>37</td>
<td>Small</td>
</tr>
<tr>
<td>23300</td>
<td>Japan</td>
<td>21</td>
<td>Medium</td>
</tr>
<tr>
<td>13800</td>
<td>Japan</td>
<td>NA</td>
<td>Sporty</td>
</tr>
<tr>
<td>27900</td>
<td>Japan</td>
<td>NA</td>
<td>Sporty</td>
</tr>
<tr>
<td>9995</td>
<td>USA</td>
<td>23</td>
<td>Compact</td>
</tr>
<tr>
<td>21498</td>
<td>Japan</td>
<td>23</td>
<td>Medium</td>
</tr>
</tbody>
</table>

we could generate the same data frame as above with the commands


> col.lab <- c("Price", "Country", "Mileage", "Type")
> some.more.data <- read.table("auto2",col.names=col.lab,row.names=row.lab)

Suppose we wanted to change the column name “Type” to “Size”

> col.lab[4] <- "Size"
> dimnames(some.more.data) <- list(row.lab,col.lab)

There are several options to the function read.table, including ones for changing the default separator from blanks to something else and for specifying a fixed format field. Refer to *Statistical Models in S* for more information.

Now, suppose we have some variables names, age, salary, sex. If each variables has the same observation index, they can be meaningfully combined in a data.frame.

> col.lab2 <- c("Name", "Age", "Salary", "Sex")
> personnel.info<- data.frame(name,age,salary,sex,col.names=col.lab2)

The existing data frame could be augmented to include another variable

>personnel.info <- data.frame(personnel.info,job.title)
If you do not specify row names and column names for your data frames, default values will be assigned. The default column names are V1, V2, V3 and so on. By default, the rows are indexed by the row number.

Suppose you have a previously defined data frame `personnel.info` and you would like to access the variables contained in that data frame individually by name (i.e. name, age, salary, job.title). Use the command

```r
> attach(personnel.info)
```

To remove the data frame from the search list, use

```r
> detach(personnel.info)
```

**For Loops, While Loops, Repeat Loops**

The structure of various looping structures can be seen in the following examples. Usually an index variable will be something like `(i in 1:100)` but it could be almost anything…

```r
for (i in c("red", "blue", "green", "yellow")) {
  if (color == i)
    print(color)
  else
    print("not available")
}
```

```r
while (stock != 0) {
  amount <- get.order()
  stock <- process.order(amount)
}
```

where `get.order()` and `process.order()` are functions

```r
repeat {
  i <- i + 1
  if (i == 100) break
}
```

where break causes abnormal termination from the loop, so it is generally better to use the for and the while constructs.

**Conditional computations:** `if, switch, &&, ||`

The general form for if

```r
if (condition)
  expr1
else
  expr2
```
Multiple expressions should be enclosed in \{ \}

For multiple conditions:

\begin{align*}
& \text{if (cond1 && cond2)} \quad \text{captures cond1 AND cond2} \\
& \text{if (cond1 || cond2)} \quad \text{captures cond1 OR cond2} \\
& \text{if (cond1 & cond2)} \quad \text{captures element-wise AND} \\
& \text{if (cond1 | cond2)} \quad \text{captures element-wise OR}
\end{align*}

Suppose we have multiple conditions all on one variable. Then we can use the “switch” command. These two segments of code do the same thing

\begin{verbatim}
if (command== “plot”) do.plot( )
else if (command== “print”) do.print( )
else stop(“command should be plot or print”) 
\end{verbatim}

\begin{verbatim}
switch(command, 
  plot = do.plot( ),
  print = do.print( ),
  stop(“Command should be plot or print”))
\end{verbatim}

**Functions**

Almost everything in R can be viewed as a function. Even when you leave R, you type q(), which is a function that takes no parameters. If you are running programs in batch mode, you can specify your functions within your program, and they will be automatically saved as functions so that you could use them in later programs. Here is a sample function to calculate the Kendall-Tau statistic (used to measure the degree of correlation between rankings of variables)

\begin{verbatim}
Kendall.tau <- function(y,z) {
  # This function calculates the Kendall-tau statistic
  # given the input vectors y and z
  # Under the null hypothesis that y and z are independent
  # the permutation distribution of tau has mean ½ and variance
  # (2n+5)/(18n(n-1))

  n <- length(y)
  sum1 <- 0

  # Put vector y in ascending order since largest element
  # does not have to go through list
  indices <- order(y)
  z <- z[indices]
  y <- y[indices]

  for (i in 1:(n-1)) {
    targ <- y[i]
    ysub <- y[y>targ]
    zsub <- z[y>targ]
    ydiff <- ysub-targ
    zdiff <- zsub-targ
    msign <- sign(ydiff)*sign(zdiff)
    sum1 <- sum1+sum(msign)
  }

  # Final calculation of the statistic
  return(sum1/(n*(n-1)/2))
}
\end{verbatim}
\[
\tau \leftarrow \frac{1}{n(n-1)} \sum_1 + 0.5
\]

\tau

Notice that the last line of the function is always the object that is returned by the function. In simple functions, brackets are not needed.

\texttt{> new.square <- function (x) + x^2}

Lines can be separated by semi-columns instead of a new line if you wish.

References


Many of the examples in this hand-out come from these two books.