```
###
###  A First Introduction to R
###
###  Author:   Stefan Evert
###  Modified: Mon Apr 18 18:24:17 2005 (evert)
###

## How to use this tutorial:
##   Lines beginning with a # character are comments, and should be read. :o)
##   All other text lines are R commands. Use cut & paste to copy them into the
##   R command window. If you are running R from Emacs/ESS, first start an R
##   process (M-x R RET). Then you can execute the current line in this file
##   by pressing C-c C-n. (Note to Emacs novices: C-c means "press the 'c' key
##   while holding down the Ctrl key"; M-x refers to the "meta" key, which is
##   often mapped to the left Alt key.)


###
###  Entering and exitting R
###

## To start an R session from the command-line, simply type "R". Windows and Mac
## users will have to click on an icon or select R from the Start menu. The most
## difficult thing in your first session will be to find the exit (no, "exit" doesn't
## work). The trick is to type:

q()

## Don't forget the parentheses! When asked whether to save the "workspace image",
## answer 'n' for no.


###
###  Emacs / ESS
###

## The best front-end to R is the ESS library in Emacs. If ESS is installed on
## your computer, you can type
##
##   M-x R RET
##
## to start an R session. NB: you can have multiple R sessions at the same time,
## so be careful not to re-enter the command above accidentally when you're already
## running an R process. The *R* buffer works just like the usual R command window.
## However, ESS adds many useful commands, most of which are accessible from the
## menu bar. Important key sequences are
##
##   C-c C-q  exit R (will ask whether you're sure and clean up)
##   M-p      move through command history (backwards)
##   M-n      move through command history (forwards)
##   M-r      search command history with regular expression (backwards)
##   TAB      complete R functions, variables, filenames, etc.
##   C-c C-e  show as much output from previous command as possible
##   C-c C-v  edit R object in separate buffer
##            (type C-c C-l in edit buffer to update object in R process)
##
## The ESS keys handout lists many other useful key sequences. ESS will also
## recognise files with the extension ".R" as R scripts and support syntax
## highlighting and automatic indentation, as well as some advanced editing functions:
##
##   C-c TAB  complete R function or variable (like TAB key in *R* buffer)
##   C-c C-l  execute entire script in R process (e.g. function definitions)
##   C-c C-n  execute current line in R process and move to next command
##            (useful for "interactive" scripts such as this tutorial)
```

```
##   C-c C-r  execute region (i.e. part of script)
##


###
###  Using R as a pocket calculator
###

## Enter basic arithmetic expressions with numbers (R does not distinguish
## between intergers and real numbers), + - * /, and ^ or ** for exponentiation.

2 + 3 * 5
(2 + 3) * 5
2 ^ 10
## compute average of seven numbers
(5 + 4 + 5 + 7 + 10 + 2 + 6) / 7

## The result of a computation can be assigned to a variable using the ->
## and <- operators (the variable is always on the pointed side of the arrow).
## In older versions of R, _ used to be a short form of <- for faster typing and
## was not allowed in variable names. This has been changed with R 2.0 but
## the ESS interface still automatically expands _ to <- while typing.
## Variable names should only use letters [A-Za-z], digits [0-9], and the
## period [.] A variable name may not start with a digit. The value of the last
## expression is automatically assigned to .Last.value (but this does not seem
## to work within ESS!)

## assign average from above to variable
average <- .Last.value
## or assing explicitly in an ESS session
average <- (5 + 4 + 5 + 7 + 10 + 2 + 6) / 7

## print variable value (the following two lines do the same)
print(average)
average
## put variable before or after expression
a <- 6
9 -> b
print(a * b)

## Common scientific functions such as sqrt(), sin(), cos(), exp(), log(), ...
## and the constant pi are built into R.

sqrt(2)
pi
sin(pi / 2)
## sin(pi) _should_ be zero ...
sin(pi)
## Euler's constant e
e <- exp(1)
print(e)
## natural (= base e) logarithm
log(e ^ 2)
## base of logarithm can be specified as second argument
log(1000, base=10)
log(64, base=2)
## short forms exist for base 10 (common log.) and base 2 (log. dualis)
log10(.0001)
log2(sqrt(2))


###
###  Help!!!
###
```

```
## Comprehensive documentation on functions and operators is built into the R
## system in the form of help pages. It can be accessed with the help() function
## or the shorter ? operator.

## ESS users: these functions do not work with C-c C-n and must be entered directly
## in the R command buffer. Cut & paste is ok.
help(sin)
?sin
## special symbols (including most operators) need to be quoted (not necessary in ESS)
help("*")
?"->"
## for an overview of built-in mathematical functions, try the following commands
?Arithmetic
?sqrt
?sin
?log
?Special

## ESS displays help pages in a separate buffer with special key bindings. Most useful
## are "l" to execute code examples line-by-line and "q" to return to the R buffer.

## Documentation is also available in HTML format with a Java-based search engine.
## The HTML documentation includes the "Introduction to R" and "R Language Definition"
## manuals, and requires the Netscape browser with Java and JavaScript enabled.
help.start()

## Most of the functionality of the Java search engine is also available on the
## command line, using the help.search() function.
help.search("Student")                  # looking for Student's t-test
help.search("", package="graphics")     # contents of package "graphics"
help.search(keyword="hplot")            # high-level plots (see HTML docs for list of
keywords)
?help.search                            # see this help page for details



###
###  Everything is an object. Well, almost.
###

## Recent versions of S, on which R is based, have introduced an object-oriented
approach.
## Although this is not always true, it is useful to think of everything in R as an
object.
## Important object types are
##
##   scalar       a single number, string, or boolean
##   vector       a vector of numbers, strings, or booleans (homogeneous)
##   list         a collection of arbitrary objects, often identified by names
##   function     a snippet of code (internally known as a "closure") which can be
##                called with optional arguments and returns an object
##   data.frame   a two-dimensional table representing the results of an experiment
##                or a sample; rows correspond to observations (or objects), and
##                columns correspond to variables (or characteristics); columns (and
##                sometimes also rows) are identified by names
##
## Objects can be named by assigning them to variables. R's kind of object-orientation,
## in which every variable knows what kind of data it holds, is largely transparent to
the user.
## It just means that functions such as print(), summary(), or plot() will almost always
## present data in an appropriate manner.
```

```
###
###  Vectors
###

## The simplest data structure in R are vectors, which are typically used to represent
## a sequence of observations. Use the c() function ("combine") to create a vector.

## vector of numbers
v <- c(1, 2, 3)
print(v)
## vector of strings
v2 <- c("Huey", "Lewey", "Dewey")
print(v2)
summary(v2)
## c() can also be used to append to a vector, or to combine multiple vectors
c(v, 8, v)

## Several special vector functions are available. length() shows the length of a
## vector, and mode() the type of data stored (numeric, character or logical).
## sum() and prod() compute the sum and product of all elements of a numeric vector,
## and mean() the average value. max() and min() find the largest and smallest value.

## to compute average of seven numbers, we will usually store the numbers in a vector
numbers <- c(5, 4, 5, 7, 10, 2, 6)
## the following lines do exactly the same
sum(numbers) / length(numbers)
mean(numbers)
## mean is one of the items in the summary of a vector
summary(numbers)

## Sequences of (equidistant) numbers can be generated with the ":" operator, or
## more generally with the seq() function.

v <- 1:10
print(v)
sum(v)
## factorial of 10 is the product 1 * 2 * ... * 10 =: 10!
prod(v)

## sequences of equidistant values are useful for tabulating and plotting functions
x <- seq(from=0, to=2, by=0.1)
print(x)


## Mathematical operators and functions take vectors as arguments, and are applied
## to each element in turn. For binary operators, vectors and scalars can be mixed.

10 * v + 3
v * v
sqrt(x)

##:: GET SOME PRACTICE: tabulate the factorials n! for n = 1, ..., 14 with a
##:: single R command. A similar command allows you to tabulate the sums
##:: s_n = 1 + 2 + ... + n  for n = 1, ..., 100.  (hint: ?cumsum)

## In order to access a single element in a vector, the element's index has to
## be given in square brackets (this is called a subscript). Indices start with
## 1 and go up to the length of the vector. You can also modify a subscripted element
## by assigning to it.

squares <- v * v
## 7 squared (should be 49 :-)
squares[7]
## perhaps _this_ is the question ... ?
```

```
squares[7] <- 42
squares

## If you want to extract several elements at once (i.e. a subsequence), you
## have to supply a vector of indices.

## squares of even numbers (note that squares[2,4,6,8,10] won't work)
squares[c(2,4,6,8,10)]

## Comparison operators (== != < <= > >=) are also applied to each element of
## a vector in turn, resulting in a vector of boolean values. In R, the two
## possible values of a boolean are written TRUE and FALSE. They can usually
## be abbreviated as T and F (be careful: those symbols may be redefined!).
## The logical operators (!=negation &=and |=or) are applied to each element
## of boolean vectors in turn.

v >= 5
(v >= 5) & (v <= 8)
## redefining truth ...
F <- TRUE
T <- FALSE
TRUE == F
## use remove() to discard unwanted objects
remove(F)
remove(T)
TRUE == F
## the sum of a boolean vector is the number of TRUE elements
idx <- (v >= 5) & (v <= 8)
sum(idx)

## Vectors can also be subscripted with a boolean vector of the same length.
## This will return all elements for which the index vector is TRUE.

v[idx]
v[squares == 81]

##:: GET MORE PRACTICE: use a boolean index vector to identify all integers
##:: from 1 through 1000 whose square ends in the same two digits as the number
##:: itself. How many of them are there? What is the pattern for third and fifth
##:: powers instead of squares?
##:: Hint: an integer ends in two 0's iff its remainder modulo 100 is 0.
##:: In R, this condition can be expressed as  (v %% 100) == 0

###
###  Defining your own functions
###

## As it was mentioned above, an R function is just a re-usable snippet of code,
## which takes zero or more parameters, and usually returns some value. If you
## print a function (e.g. by typing just the function name), R will show you the
## actual code without running it. In order to execute the code, you have to
## specify a parameter list in parentheses (even if the list is empty, as for the
## ls() function).

## most of the functions we have encountered so far are internal to the R program
sin
sqrt
## ls() lists all variables in your workspace; this function is written in the R language
ls
ls()

## You can define your own functions with the "function" keyword, followed by a
## list of argument names (with optional default values), followed by the body
## of the function. The resulting function object is usually assigned to a variable,
```

```
## thus giving the function a name. In the simplest case, the body of the function
## is just an R expression, which yields the return value when the parameters are
## substituted. Note that this kind of function definition must be written on a single
## line.

## our function sqr() computes the square of a number x
sqr <- function (x) x * x
sqr(5)
## parameters can be passed by name (useful for functions with many parameters)
sqr(x=2)
## since the x * x is valid for vectors, we can pass a vector to sqr()
sqr(v)

## fac() computes the factorial n! using prod() as shown previously
fac <- function (n) prod(1:n)
fac(5)
fac(12)
## since the expression used to calculate the factorial only works for scalars,
## we cannot pass a vector to the function; R does not automatically apply it
## to each element in turn as some other programs do (called "threading")
fac(v)
## however, we can explicitly apply the function to individual elements with sapply()
sapply(v, fac)

## Function arguments are optional if they have a default value. This is often
## used for "parameters" which change the behaviour of the function.

## compute the p-norm of a vector; for p=1 this is the sum of (the absolute values
## of) the elements, for p=2 we get the Euclidean norm (default)
norm <- function (v, p=2) (sum(abs(v) ^ p)) ^ (1/p)
## two ways of computing the Euclidean norm
norm(v)
norm(v, p=2)
## L1-norm and L4-norm of the vector v
norm(v, p=1)
norm(v, p=4)


###
###  Graphics: simple plots.
###

## R provides a wide range of high-quality plotting functions. Screen graphics
## can easily be saved to EPS files for printing. A simple type of plot which
## is useful for a one-dimensional list of numbers is the histrogram-like barplot.

## the argument supplied to barplot is a vector of numbers
barplot(sqrt(v))
## we can label the y-axis with the ylab argument, print a title above the plot (main),
## and supply legends for the individual bars with names.arg (as a vector of strings)
barplot(sqrt(v), ylab="square root", main="Sample Plot", names.arg=v)
## note that we can pass the numeric vector v as names.arg; R will automatically
## convert the numbers into strings

## Plots can be fine-tuned with the par() function, which sets various graphics
parameters.
## For instance, the "cex" parameter can be used to enlarge letters and symbols.
par(cex=1.3)                            # default size is 1.0
barplot(sqrt(v), ylab="square root", main="Sample Plot", names.arg=v)

##:: PLOTTING PRACTICE: plot the L1-, L2-, ..., L8-norm for our vector v using barplot().
##:: The bars should be labelled with the name of each norm ("L1", "L2", ...).
##:: Can you compute all 8 norms in a single line? (hint: think of the sapply() function)
```

```
## The most common type of plot in R is the scatter plot, for which the X and Y
## coordinates of individual points are specified. We can also use scatter plots to
## draw graphs of mathematical functions. As an example, we will plot the graphs
## y = x^2, y = x, and y = sqrt(x) for 0 <= x <= 2. We have to set the coordinate ranges
## of both axes explicitly (with the xlim and ylim parameters) to ensure that they
## use an identical scale; xlim and ylim are given as two-element vectors.
## The function points() allows us to draw multiple graphs in a single scatter plot.

## choose equidistant X coordinates for the plot, then compute the corresponding Y coords
x <- seq(from=0, to=2, by=0.05)
y1 <- x ^ 2
y2 <- x
y3 <- sqrt(x)

## plot the first function (scatter plot style with small circles)
plot(x, y1, xlim=c(0,2), ylim=c(0,2), main="Some functions", xlab="X", ylab="Y")
## "line"-type plots give the impression of a continuous function
plot(x, y1, xlim=c(0,2), ylim=c(0,2), main="Some functions", xlab="X", ylab="Y",
type="l")
## add graphs to the plot with points(), varying colour with the col parameter
points(x, y2, type="l", col="red")
points(x, y3, type="l", col="blue")

## legends are a little tricky ... (the lwd parameter stands for "line width")
legend(x=0, y=2, legend=c("y=x^2", "y=x", "y=sqrt(x)"), col=c("black", "red", "blue"),
lwd=c(2,2,2))

## use (symbolic) expressions to set mathematical formulae, but we have to redo the plot
plot(x, y1, xlim=c(0,2), ylim=c(0,2), main="Some functions", xlab="X", ylab="Y",
type="l")
points(x, y2, type="l", col="red")
points(x, y3, type="l", col="blue")
leg.labels <- expression(y==x^2, y==x, y==sqrt(x))
legend(x=0, y=2, legend=leg.labels, col=c("black", "red", "blue"), lwd=c(2,2,2))

## save plot to file (encapsulated postscript format)
dev.copy2eps(file="myplot.eps")

## Natural-language data are often spread out across wide frequency or parameter ranges.
## Logarithmic plots help to fit the full range into a single display, and make it easier
## to identify exponential and power-law relationships between variables. The log
parameter
## specifies whether one or both axes are to be scaled logarithmically. The plotting
range
## for a logarithmic axis must be positive. First, let us see how (single) logarithmic
plots
## transform exponential and logarithmic functions (on the interval [0, 4]).

x <- seq(0.01, 4, by=0.01)
y1 <- log2(x)
y2 <- x
y3 <- 2 ^ x
leg.labels <- expression(y==log2(x), y==x, y==2^x)
leg.col <- c("red", "black", "blue")

## non-logarithmic scale
plot(x, y2, xlim=c(0, 4), ylim=c(-1, 3), xlab="X", ylab="Y", type="l")
points(x, y1, type="l", col="red")
points(x, y3, type="l", col="blue")
abline(h=0, lty="11")
abline(v=0, lty="11")
legend(x=0, y=3, legend=leg.labels, col=leg.col, lwd=2)

## on an x-logarithmic scale, logarithmic functions become straight lines
```

```
plot(x, y2, log="x", xlim=c(0.5, 4), ylim=c(-1, 3), xlab="X", ylab="Y", type="l")
points(x, y1, type="l", col="red")
points(x, y3, type="l", col="blue")
abline(h=0, lty="11")
legend(x=0.5, y=3, legend=leg.labels, col=leg.col, lwd=2)

## on a y-logarithmic scale, exponential functions become straight lines
plot(x, y2, log="y", xlim=c(0, 4), ylim=c(0.5, 4.5), xlab="X", ylab="Y", type="l")
points(x, y1, type="l", col="red")
points(x, y3, type="l", col="blue")
abline(v=0, lty="11")
legend(x=0, y=4, legend=leg.labels, col=leg.col, lwd=2)

## on a double-logarithmic scale, all power functions (incl. y=x) are straight lines
y1 <- sqrt(x)
y2 <- x
y3 <- x ^ 4
leg.labels <- expression(y==sqrt(x), y==x, y==x^4)
leg.col <- c("red", "black", "blue")
## "normal scale": only y=x is a straight line
plot(x, y2, xlim=c(0, 4), ylim=c(0, 4), xlab="X", ylab="Y", type="l")
points(x, y1, type="l", col="red")
points(x, y3, type="l", col="blue")
abline(h=0, lty="11")
abline(v=0, lty="11")
legend(x=0, y=4, legend=leg.labels, col=leg.col, lwd=2)
## double-logarithmic scale: slope of line is equal to exponent of power function
plot(x, y2, log="xy", xlim=c(0.1, 4), ylim=c(0.1, 4), xlab="X", ylab="Y", type="l")
points(x, y1, type="l", col="red")
points(x, y3, type="l", col="blue")
legend(x=0, y=4, legend=leg.labels, col=leg.col, lwd=2)


###
###  Observation tables: data frames
###

## The most important object type in R are data frames. They represent the two-
dimensional
## tables that result from a statistical sample or an experiment. Each row corresponds to
## a single observation (i.e. an object in the case of a random sample). Each column
lists
## the observed values of a variable (i.e. one characteristic of the objects in the
sample).
## Columns are identified by the name of the corresponding variable, whereas rows are
numbered
## starting with one. Rows are often labelled as well (e.g. with the name of a subject,
or
## the number/id of an observation). R includes a collection of built-in data sets, which
are
## useful for examples and for playing around.

## list built-in data sets and get further information about them
data()
?mtcars

## load built-in data set: will be available under the specified name
data(mtcars)
print(mtcars)

## use dim() to find out size of data set; returns vector (<rows>, <columns>)
dim(mtcars)

## labels of columns (variables) and rows (objects)
```

```
colnames(mtcars)
rownames(mtcars)

## R's simple data editor can be convenient for viewing data sets, too;
## be careful not to edit cells accidentally (press ESC to restore cell)
fix(mtcars)

## for data frame objects, summary() prints a summary of each variable
summary(mtcars)

## plotting a data frame produces a matrix of scatter plots (showing each pair
## of variables), which may help to pick out correlations visually
plot(mtcars)

## Subsetting data frames works similar to vector subscripts (but note the
## extra comma to extract entire rows); multiple lines can be select with
## a vector of line numbers, or a boolean vector of appropriate size. There
## is a special syntax for accessing single columns.

## show the tenth row, and the entry for the Maserati Bora
mtcars[10,]
mtcars["Maserati Bora",]

## show miles per gallon, displacement and horsepower for first five cars
mtcars[1:5, c("mpg", "disp", "hp")]

## short syntax for column: number of cylindes for all cars in the data set
mtcars$cyl

## show all cars with 8 cylinders (using a boolean index vector)
mtcars[(mtcars$cyl == 8), ]

## We will learn later how to load and save data frames so that we can create
## and analyse our own data sets.
```