R quick guide

```
q()                             quit
help(function), ?function       get help
??keyword                       help search
```

Basic elements

Variables

Variable names (identifiers), can be any sequence of letters a-z, A-Z, digits 0-9, underscore _, or point ., but cannot start with a digit and should not start with a point (which has a special meaning).
Examples: `my_name, varible_1, name.with.points, Anna, Charlie.`
Variable assignment is done with the <- operator (or =):

```
my_first_variable <- 10
my_second_variable = 50
var.3 <- my_first_variable + 3.14
```

Vectors

Indexed values of the same *mode* (see below)

```
c(3, 4.5, 6, 78)          combine values to a vector
x <- c(4, 5, 6)
y <- c(x, 45, x)
```

vector arithmetic

```
z <- 2*x + 3
x2  <- x^2
```

operators: `+, -, *, /, ^`

sequencies

```
1:30
30:1
seq( a, b, by=step)
seq(1, 2, by=0.1)                    1.0, 1.1, 1.2,... 2.0
seq(length=5, from = 2, by = 0.5)    2.0, 2.5, 3.0, 3.5, 4.0
seq( from=10, by=2, along=y)         10, 12, the same length as y
seq(along=y)                         1, 2, 3, 4, 5 … length of y
```

repetitions

```
rep(1:4, 2)                    1 2 3 4 1 2 3 4
rep(1:4, each = 2)             1 1 2 2 3 3 4 4
rep(1:4, c(2,1,2,1))           1 1 2 3 3 4
rep(1:4, each = 2, len = 10)   1 1 2 2 3 3 4 4 1 1
rep(1:4, each = 2, times = 3)  1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2
2 3 3 4 4
```

logical vectors

```
x <- 1:10
x < 5
```

```
TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
y <- c(T, F, F, T, T)   TRUE FALSE FALSE  TRUE  TRUE
```

operators: <, >, ==, |, &,
T and F are predefined values. Don't use T or F as variable names!

Indexing

```
x <- seq(2, 10, by=2)   2 4 6 8 10
x[2]                    4
x[3:4]                  6 8
x[-3]                   2 4 8 10
x[-(1:3)]               8 10
x[x>5]                  6 8 10
x[x>5] <- 0             2 4 0 0 0
```

Characters and strings

Strings are 'character objects'. A variable with a string value is treated as a single object. Use `substr` to extract parts of it.

```
s <- "hello"
substr(s, 3, 5)         "llo"
```

Special values

```
T             TRUE
F             FALSE
NA            Missing value
NaN           Not a number (0/0 = NaN, Inf - Inf = NaN)
Inf           infinity ( 1/0 = Inf , -1/0 = -Inf)
is.na(NA)     TRUE
is.na(1.0)    FALSE
is.na(NaN)    TRUE
is.na(Inf)    FALSE
is.nan(NA)    FALSE
is.nan(1.0)   FALSE
is.nan(NaN)   TRUE
is.nan(Inf)   FALSE
```

Lists

Lists are like vectors, an indexed set of objects (called *components*), but the objects need not be of the same type. Each component can be named. Components can be extracted through the `$` operator or by [[double brackets]].

```
> mylist <- list(car="volvo", weight=9, "unnamed")
> mylist
$car
[1] "volvo"

$weight
[1] 9
```

```
[[3]]
[1] "unnamed"

> mylist$car
[1] "volvo"
> mylist[[1]]
[1] "volvo"
> mylist[[3]]
[1] "unnamed"

> y <- list(1:5)
> y
[[1]]
[1] 1 2 3 4 5

> y[[1]]
[1] 1 2 3 4 5
```

Atributes of objects

mode — atomic: numeric, complex, logical, character, raw; recursive: list, function, expression

```
s <- "hello"
mode(s)                 "character"
as.character(23)        "23"
as.integer("43")        43
```

length — number of elements

```
x <- 2:5                2 3 4 5
length(x)               4
length(x) <- 2          2 3
length(x) <- 4          2 3 NA NA
```

Note: a string has *length* 1 (it is a single character object):

```
length("hello")         1
```

To find out the number of characters in a string use nchar():

```
nchar("hello")          5
```

names — may be used as extra description

```
nx <- 3
names(nx) <- "Number of items"
nx                      Number of items
                                      3
unname(nx)              removes the name
```

Notice: Names are inherited! Try:

```
ny <- 2*nx
ny                      Number of items
                                      6
```

Arrays and Matrices

Arrays can be of any dimensionality. Matrices are two-dimensional arrays.
Dimensions are set through the dim attribute.

```
x <- 1:9                    This is a vector
dim(x) <- c(3,3)            Now it's a matrix
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

indexing:

```
x[3,2]                      6
x[3,1:2]                    3 6   (it's a vector!)
x[,3]                       7 8 9 (it's a vector!)
```

arrays

```
> x <- array(1:12,dim=c(3,4))
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

or simply:

```
> x <- array(1:12,dim=c(3,4))
```

index matrices

Each row is a set of indices, specifying a single element.

```
> ii <- array(c(1,1,3,4,2,1), dim=c(3,2))
> ii
     [,1] [,2]
[1,]    1    4
[2,]    1    2
[3,]    3    1
> x[ii]
[1] 10  4  3
> x[ii] = 5
> x
     [,1] [,2] [,3] [,4]
[1,]    1    5    7    5
[2,]    2    5    8   11
[3,]    5    6    9   12
```

Recycling

Vectors in expressions with larger vectors or arrays are recycled to fit the format.

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> y <- x + c(4,1)
Warning message:
```

```
In x + c(4, 1) :
  longer object length is not a multiple of shorter
object length
> y
[1] 5 3 7 5 9
```

We got a warning, but a result was still produced. If the number of elements in the larger sized object is a multiple of the length of the shorter vector, there is no warning.

```
> A = array(1:6,c(2,3))
> A
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> B = A + 3
> B
     [,1] [,2] [,3]
[1,]    4    6    8
[2,]    5    7    9
> B = A + c(3,4)
> B
     [,1] [,2] [,3]
[1,]    4    6    8
[2,]    6    8   10
> B = A + c(3,4) + c(0,1,0)
> B
     [,1] [,2] [,3]
[1,]    4    6    9
[2,]    7    8   10
> B = A + c(3,4) + c(0,1,0) + 1:5
Warning message:
In A + c(3, 4) + c(0, 1, 0) + 1:5 :
  longer object length is not a multiple of shorter
object length
```

Note the difference between arrays, that have dimensions, and vectors, that are just a sequence of objects. R recycles vectors, but not arrays.

```
A <- array(0,c(2,3))
> A
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> B <- A + c(0,1,0)
> B
     [,1] [,2] [,3]
[1,]    0    0    1
[2,]    1    0    0
> B <- A + as.array(c(0,1,0))
Error in A + as.array(c(0, 1, 0)) : non-conformable
arrays
> dim(c(0,1,0)) # this is a vector
NULL
> dim(as.array(c(0,1,0))) # this is a one-dimensional
array
[1] 3
>
```

Generating random numbers

R can generate random numbers from a whole suite of distributions. The most useful are the uniform distribution, `unif`, and the normal distribution, `norm`. The commands `runif` and `rnorm` can be used to generate random numbers. Each call generates a new number.

The uniform distribution has two parameters, min and max, which default to 0 and 1.

```
> runif(1)
[1] 0.3115796
> runif(3)
[1] 0.7157580 0.0424130 0.6362725
> runif(3,min=5,max=9)
[1] 7.496739 8.559932 8.148903
```

The normal distribution has two parameters, mean and sd, with default values 0 and 1.

```
> rnorm(5)
[1]  0.28217973  0.06348225 -0.49138284  0.41875925 -
0.32267752
> rnorm(5,mean=100,sd=10)
[1] 114.59594 115.76307  99.39165  96.17111 103.80130
```

The sample functions are also useful:

```
sample(v,k) # samples k elements from vector v
sample.int(n,k) # samples k numbers (integers) from 1:n
```

```
sample(v,k,prob=weights) # weighted sampling
sample.int(n,k,prob=weights) # weighted sampling
```

Program control (*really* brief, please see lecture notes)

Commands and compound commands

Commands are separated by ; or newline
compound, or grouped, commands are grouped by {}. The value of the group, if required, is the value following a `return` statement or, if `return` is missing, the value of the last command.
comments start with #

command history by arrow keys

```
source("commands.R")    execute script commands.R
sink("output.txt")      divert output to output.txt

objects()               see all objects in workspace
ls()                    - " -
rm(x)                   remove object (delete variable)
rm(list=ls())           remove all objects (clear
                        workspace)
```

If statements

Syntax:

`if (`*`expression1`*`)` *`expression2`* `else` *`expression3`*

All expressions can be compound, i.e. expressions grouped with {curly brackets}. expression1 must evaluate to a logical value (TRUE or FALSE). The else is not compulsory.

For loops

Syntax:

`for (`*`name`* `in` *`expr1`*`)` *`expr2`*

Repeat

`repeat` *`expression`*
will repeat *`expression`* indefinitely until a *`break`* statement is encountered.

While

`while (`*`condition`*`)` *`expression`*
repeats *`expression`* as long as *`condition`* is true (can be zero times).

Functions

Definition:

`name <- function(`*`arg_1, arg_2, ...`*`)` *`expression`*

Usage:

```
name(arg_1, arg_2, ...)
```
or
```
myvar <- name(arg_1, arg_2, ...)
```

The value of the function is simply the value of *expression*.

Scope

Variables can not be accessed from anywhere. Luckily. As a general rule, variables can only be used within the *scope* or *environment* in which they are created. Functions have their own scope, so variables defined within a function are not accessible from outside the function. A notable exception to this rule in R is that functions can access variables defined in higher levels.

Other handy functions:

```
cat
length, ncol, nrow
sort, order
sample
rbind, cbind
sum, cumsum
rowSums, colSums, rowMeans, colMeans
round, floor, ceiling
mean, median, min, max, range, var, sd
abline
```