The R language – a short companion

This companion is essentially based on the documents „An Introduction to R“ and „R language definition“, both version 1.7.1, available on the R website http://www.r-project.org/. Graphical and statistical functionalities are not considered.
Version 1.2. Marc Vandemeulebroecke, July 14th, 2003

Objects in R

- are referred to through **symbols** (see below)
- have **type, mode, storage mode**:
  - **types:** *NULL, symbol, pairlist, closure, environment, promise, language, special, builtin, logical* (e.g. NA[1])*, integer, double* (e.g. NaN[2] or Inf)*, complex, character, ‚...‘, any, expression, list, externalptr* and *weakref*
  - **modes:** basic nature of the object’s fundamental constituents: *numeric, complex, logical, character, list, function, call, expression, name,* etc
  - **storage modes:** *logical, integer, double, complex, character, symbol* etc
- most important objects (these are referred to through variables):
  - **vectors, arrays, matrices:** Vectors are an *atomic structure*,and may be of the types *logical*, *integer*, *double*, *complex* and *character*, with certain associated modes and storage modes[3]. Scalars are treated as vectors of length 1. **Arrays** are vectors plus the dim attribute (dimension vector), **matrices** are arrays with a dim attribute of length 2. Arrays are ordered column major order. See also „Indexing“ below.
  - **factors:** handle nominal and ordered categorical data. Have a levels (and optionally a contrasts) attribute and class *factor* (note the effect of unclass()!). Implemented as an integer vector and a mapped vector of names for the levels. typeof() returns *integer*, mode() returns *numeric*. The combination vector + labelling factor is an example of a *ragged array*.
  - **lists:** *recursive structure*. „Generic vectors“, of type and mode *list*; their *components* (top-level elements) can be of any type or mode. See also „Indexing“ below.
  - **data frames:** matrix-like structures with columns of possibly different type/mode/attributes; comparable in form and function to SAS datasets. More precisely: lists of class *data.frame* (note the effect of unclass()!), built of (numeric, character[4] or logical) vectors, factors, (numeric) matrices, lists and/or data frames of the same length[5]. The columns are implemented as list components. Data frames usually have a names attribute for the variables (columns) and a row.names attribute for the cases (rows). See also „Indexing“ below.
  - **functions / function closures:** *recursive structure*. Have basic components *formal argument list* (symbols, ‚symbol=default‘-constructs, ‚...‘), *body* and *environment* (→ lexical scoping). Are of type *closure* and mode *function*. May be assigned to symbols or anonymous. May also serve as function arguments or function return values, e.g.: g <- function(f) function(v) sum(f(v)).
- other objects:
  - **calls:** sometimes referred to as „unevaluated expressions“; are of mode *call*

[1] *logical* is NA’s default type, it may get coerced to other types.
[2] NaN⇒NA
[3] see appendix
[4] coerced into factors
[5] no. of rows for the matrices and data frames, component length for the lists

  - **expressions:** *recursive structure*. Are of mode *expression*. (*statements* = syntactically correct expressions)
  - **names:** are of mode *name*.

  These three objects constitute as *language objects* the R language.
  - **symbols:** (names of) R objects. Are of type *symbol* and mode *name*.
  - **NULL:** marking a missing object. Has no modifyable properties.
  - **objects of type *expression*:** contain parsed but unevaluated R statements
  - **objects of type *builtin* or *special*:** built-in (".Primitive()") functions, e.g. c()
  - **objects of type *promise*:** used for R's *lazy evaluation* of function arguments
  - **objects of type „...':** „...' is used for unspecified formal function arguments.
  - **objects of type *environment*:** consist of a *frame* (set of symbol-value pairs) and a pointer to an enclosing environment, e.g. *.GlobalEnv*
  - **objects of type *pairlist*:** something internal
  - **objects of type *any*:** rarely used
- have **attributes** (mostly as vectors; user-defined attributes may be added), e.g.:
  - **mode:** see above. *intrinsic attribute*
  - **length:** number of (highest level) elements[6]. *intrinsic attribute*
  - **dim:** used to implement arrays[7]
  - **names:** used to label the elements of a vector (/array) or list, or the variables of a data frame. (For one-dimensional arrays, dimnames[[1]] is accessed.)
  - **dimnames:** list of character vectors used to label the dimensions of an array.
  - **class:** vector of character strings („class names") used for OO programming[8], e.g. *data.frame*, *c("ordered", "factor")* or *table*
  - **tsp:** used for periodic time series
  - **levels:** for factors; of type *character*
  - **contrasts:** for factors
  - **row.names:** to label the cases of a data frame
  - **rownames/colnames:** first dimnames; = row.names/names in a data frame
  - **source:** the source code of user defined functions
  - **comment:** is handled by the function comment()
- Some functions in this context:
  - typeof(), mode(), storage.mode(), attributes(), attr(), length(), dim(), names(), dimnames(), class(), tsp(), levels(), contrasts(), rownames(), row.names(), colnames(), mostly returning a vector (a list for attributes()); comment()
  - vector(), matrix(), array(), factor(), ordered(), list(), data.frame(), function(), call(), expression() (note the difference to as.expression()), creating the respective object
  - is.object(), is.vector(), is.array(), is.matrix(), is.factor(), is.ordered(), is.list(), is.data.frame(), is.function(), is.language(), is.call(), is.expression(), is.symbol() (=is.name()), is.null(), is.pairlist(), is.environment(), is.table(), is.ts(), is.tskernel(), is.atomic(), is.recursive(), is.logical(), is.integer(), is.double(), is.real(), is.complex(), is.character(), is.numeric(), is.na() (vectorized), is.nan() (vect.), is.finite (vect.), is.infinite (vect.), returning logical vectors (mostly of length 1)
  - as.vector(), as.symbol() (=as.name()), as.numeric(), as.integer() etc, attempting a coercion of objects/modes/types etc; unlist(). E.g.: x <- 1; eval(as.symbol("x")).

[6] For data frames, this means number of columns (see above).
[7] dim works also for lists, but is rarely used there. Try l <- list(1, 2); ll <- list(l, l); dim(ll) <- c(2,1) (Not list(l, l, dim=c(2, 1))!).
[8] class() returns the *implicit class* ("matrix", "array" or the result of mode()) if an object has no class attribute.

Indexing

- Index operators:
  - **[**, e.g. x[2,1,1]: General subscripting operator, selects top-level elements.
  - **[[**, e.g. x[[2,1,1]]: Rarely used for vectors and matrices. Allows selection only of one single element, otherwise similar to [, except (e.g.) for the attributes handling: [[drops names and dimnames. See below for details.
  - **$**, e.g. x$a or x$"a": Selects a (named) list component by its *list tag*.

  All forms extract or replace subsets (e.g., may be used on an assignment's left side[9]). They are nothing but functions, or more precisely *subset operators*. Even more precisely: **[[accesses single elements down the hierarchy, [accesses subsets on the top level.** The indices of [and [[may be computed using expressions, e.g. x[1+1].
- Indexing vectors with [:
  - Indexing a vector by an ***index vector*** returns a vector of the specified elements. E.g., (3:1)[rep(1, times=2)] returns the vector c(3,3). The index vector may be of type *integer* (all positive or all negative, selecting or deselecting elements; 0 has no effect), *logical* (TRUE selects; recycling rules apply!), *character* (selecting elements with matching names), or other numeric types (truncated to *integer*). For an indexing factor, the integers of the underlying array are used, not the levels.
  - x[] = x (but dropping „irrelevant" attributes).

  Indexing with [preserves names.
- Indexing arrays with [(using x <- array(8:1, dim=c(2,2,2)) as an example):
  - A single element may be read into a vector of length 1 by addressing its position in the array, e.g. x[1,2,1]; names (and of course dim and dimnames) are dropped. One empty index position results in a vector; names (and of course dim) are dropped, dimnames converted into names. More empty positions result in a subarray; names are dropped, dimnames preserved and dim possibly reduced. Any index position may more generally be filled with an index vector in the above sense, e.g. x[TRUE,c(1,2),], returning a vector (possibly of length 1) or a subarray, with the same handling of attributes. An empty index position is equivalent to the index 1:dim(x)[indexposition].
  - A single index vector in the above sense accesses the *data vector* of the array and returns a vector; names are preserved. E.g., x[5]. x[1:length(x)] is the data vector.
  - x[] = x as for vectors (preserving dim, dimnames and names, unlike x[,,]).
  - One or more elements may be read into a vector by an ***index matrix*** with one row per addressed element. E.g., x[matrix(c(1,2,1), nrow=1, byrow=TRUE)] returns the same result as x[1,2,1]. names are dropped.

  0 „drops out" of an index vector as described above. A single 0 in an index postion returns an empty structure; names are dropped, dim is reduced and dimnames behave in a complicated way. x[0] returns named numeric(0).
  drop=FALSE prevents the reducing of dim or coercion into a vector.
- Indexing vectors or arrays by [with NA gives NA results; the exact behaviour depends on NA's type.
- which() finds the "TRUE" indices of vectors or matrices, e.g.: which(3:1<2).
- Indexing lists (using lst <- list(age=3:1) as an example):

[9] This also enables the extension of a vector or list, frequently starting from an empty object. E.g., l <- list(); l$"a" <- "test". For arrays, this does not work.

List components are always numbered and may be named (by *list tags*, implemented as names attribute). They may be accessed by constructs like lst[[1]], lst[["age"]], lst$age or lst$"age"; an incomplete list tag is allowed if its completion is unambigous. E.g., b <- "a"; lst[[b]] is OK[10]. A [construct may also be used (as above), often appended to a [[or $ constuct to make a selection within a list component, e.g. lst$"a"[2:3]. For lists, [[selects any single (top-level) element (dropping names), whereas [returns a list of the selected (top-level) element(s) (preserving names); an indexing vector in [[digs deeper down the hierarchy[11]. Compare, e.g., lst["age"] to lst[["age"]]. Complex nesting is possible; list components may be given through variables. E.g., for b <- "a"; lst2 <- list(lst, b), lst2[[1]][1]$age[1] returns[12] 3, and lst2[[2]][1] returns "a". Compare also the different effects of list(3:1), list(3, 2, 1), list(age=3:1) and list(age<-3:1)!

- Indexing data frames and working with data frames (see also subset() and merge()): As data frames are lists, indexing works just as well. The function attach() may be used to place a data frame's variables at position 2 (default) in the *search path* (see also below). (Thus, they may be hidden if position 1 is filled!). detach() should be used afterwards[13]. This functionality enables some intuitive „working hygiene".

Scope and function evaluation

- *Scoping* is the internal search for the value of a symbol, most interestingly in the context of functions. In the body of a function, variables are either supplied through the formal argument list, local variables (both are *bound*) or *unbound* (=*free*)[14]. Any variable appearing in the function body is first searched for in the function's *evaluation environment*, then in its *defintion environment* (which has become the *parent environment* (*enclosure*) of the former by the call of the function) and so on up to the *global environment* (=root of the workspace), then along the *search path* of environments until the base package. (In R everything lives in (possibly nested) *environments*, which are a nesting of *frames* (=set of local variables created in a function). search() shows the search path.) Bound variables are already found on the lowest level; supplied arguments are practically treated as local. Unbound variables are not found on the lowest level. The mentioned nesting of the evaluation environment in the definition environment makes R's scoping *lexical*: The value of an unbound variable is essentially determined by the bindings that were in effect at the time of the creation of the function. In contrast, with *static* scoping as in S, it is determined on the global environment level. (*Dynamic* scoping lets it be determined by the most recent (in time) definition, jumping up the *call stack*. This is possible in R through special functions beginning with „sys.".) Some examples for R's scoping behaviour:
  - a <- 1; f <- function(x) {x+a}; a <- 10; f(0) returns 10.
  - cube <- function(x) {sq <- function() x*x; x*sq()}; cube(2) returns 8.
  - f <- function(x) {y <- 10; g <- function(x) x+y; return(g)}; h <- f(); h(3) returns 13.
- Handling of function arguments:

[10] lstb, lst"b", lst$1, lst[[age]] or lst[age] do not work here!
[11] lst[[1]][[1]] and lst[[c(1,1)]] are equivalent. At the bottom level, [[digs no further: single index selection behaves as with [, vector digging produces an error (compare lst[[1]][[1]][[1]], lst[[1]][[1]][1] and lst[[c(1,1,1)]]).
[12] This is cumbersome for the equivalent lst2[[1]]$age[1]
[13] In a different context, attach() and detach() may also take, e.g., a directory name as argument.
[14] In a <- 1; f <- function() {print(a); a <- 2; a}; f(); a, the free variable a becomes local and does not change globally.

- *Argument matching*: Formal arguments are matched to supplied arguments first by *exact matching on tags*, then by *partial matching on tags*, and finally by *positional matching*. E.g., for f <- function(aa, bb, cc) aa+bb^cc, f(2, 3, aa=1) returns 9. For f <- function(fun,fon) <body>, f(f=1, fo=2) is illegal, whereas f(f=1, fon=2) is OK. The unspecified argument ‚...' absorbs any non-matched supplied argument and is often used to pass on arguments to other functions; using it in the formal argument list before the last position may cause matching problems.
- *Argument evaluation*: Default values may be defined for formal function arguments using ‚symbol=default'[15]. Defaults may be arbitrary expressions, even involving other arguments of the same function. (missing() provides an alternative for setting defaults.) Supplied and default arguments are treated differently. R implements *call-by-value* and *lazy evaluation*, meaning that arguments are not evaluated until needed. Thus, using side-effects programming in function arguments (like in f(x <- y)) is bad style (but: x[i <- 1] is nice). (The expression used as an argument is only stored in a *slot* of a *promise*, together with a pointer to the environment the function was called from (in another slot). When (if) needed, the expression is evaluated (sensitive to any changes of the environment pointed to) and stored in yet another *slot* (avoiding a second evaluation). This is called *forcing* the promise. If a default expression is accessed, the pointer points to the function's local environment.)

Diverse topics

- R is an *interpreted* (not a *compiled*) language. For batch processing, programs should be edited in a text editor and cut and pasted, or saved and „sourced" using source(p) after setting p <- "<path to program>". Functions for help and session management include ? (=help()), help.start(), help.search(), index.search(), example(), apropos(), demo(), q() (=quit()), ls() (=objects()), rm() (=remove()), source(), sink(), save(), save.image(), load(), library() etc. Sessions may be customized[16] by the file Rprofile in the subfolder *etc* of R's home directory (valid for all sessions), the file .Rprofile in a working directory, the .RData *workspace image*[17] in a working directory and the function .First() (defined in either of the two profile files)[18] – e.g., source() or library() commands could be included here. .Last() is executed at the very session end. Data may be im- or exported by read.table() and write.table() (for data frames), scan(), data(), write() and other functions (e.g. for SAS datasets). edit() edits data in a spreadsheet style, e.g. new <- edit(data.frame()).
- *Operators* like ‚:', ‚–', ‚>=' or ‚<-', indexing constructs (see above) and even ‚{' are nothing but functions. Operators follow a certain *order of precedence*[19].
- R is *case sensitive*. ‚.' is used instead of ‚_'; alphanumeric symbols may not start with a digit, or ‚.' followed by a digit. Commands are separated by ‚;' or a newline, and grouped by ‚{}'. A comment starts with ‚#' and lasts until the end of the line, it may not be placed inside strings or a function's formal argument list. Character constants are delimited by ‚'' or ‚"', escape characters start with ‚\', e.g. \n, \t, \b. „Strange

[15] Here, sometimes a vector shows the possible argument values, with its first (scalar) entry being the default.
[16] in Windows installations – this may be slightly different under UNIX
[17] .RData and the log file .Rhistory are created when saving the session. At the next saving, .Rhistory gets appended.
[18] in this order of execution
[19] see appendix

names“ may be used for list tags etc when given as text strings; this is not possible for function arguments. Infix operators may be defined as %operator%. Note the effect of “x“ <- 1; “x“; x! Careful: TRUE, if, function, ..., c, t, %*% etc can be overwritten!

- To show an object or its properties, it may be printed using print() or just its name[20], structure(), dput() or show() (which needs a class attribute). (quote() just quotes, see below.) str() compactly shows an object’s structure. A user-defined function info() (to be included in Rprofile) may aggregate the functions typeof(), mode(), storage.mode(), length(), attributes(), and the is. family (see above) to return an alternative picture in list form. A function no.dimnames() may be defined to show arrays without dimnames.
- *Coercion* (of objects and/or types, modes...) is frequently and generally sensibly performed. E.g., when coercing numerics into logicals, the following happens: NA→NA, NaN→NA, 0→FALSE, anything else→TRUE. And vice versa: NA→NA, FALSE→0, TRUE→1. As an application, sum(1:8>7) returns 1.
- Array/Vector arithmetic:
  - Simple forms of vector/array creation are, e.g., 3, c(3,2), 3:1, c(a=“b“). More flexible is to use seq(), rep(), or some of the above mentioned functions like matrix(), as.vector() etc. Recycling may be performed, e.g. array(0, dim=2:3). Logical vectors may be created by conditions, e.g. 1:4>2.
  - Many array/vector operations are performed *elementwise* (from left to right), they are *vectorized* on the data vector. Arrays must have the same dim attribute. Vectors must be of no greater length than arrays. Short vectors are *recycled*. The result is an array with the common dim attribute of its array operands, or – if no array is present – a vector of the size of the longest vector operand. E.g., 2*x.
  - Important functions in this context are: The operators, order(), sort(), rev(), outer() (very powerful, letting, e.g., functions be evaluated on a grid), aperm(), t() (note the result of t(1:3)), cut(), diag() (with various meanings depending on the argument), solve() (also for matrix inversion), cbind() and rbind() (binding together vectors and/or matrices[21] by columns or rows (and possibly recycling shorter vectors), always returning a matrix, e.g. rbind(1:2)). The *concatenation function* c() removes any dim and dimnames attribute[22], it may also take lists.
- Control structures: if()-else (vectorized version: ifelse()), switch() and the looping statements repeat, while() and for() (and next and break) are available. However, a whole object view should be preferred to explicit looping if possible. Functions like the powerful apply() family perform implicit looping, and many operations are vectorized. any() and all() can be helpful. For switch(), eval(x[[*condition*]]) may be an alternative.
- (Frequency) *tables* may be calculated from equal length factors by table(). They are arrays with dim and dimnames attribute and class *table*. The dimension vector is built of the lengths of levels vector of each factor; the factor names and levels are taken for the dimnames list.
- Object-oriented (OO) programming[23]: R uses the ***class*** concept, ***methods*** dispatching on classes, and the idea of ***inheritance***. The most important applications are the print, summary and plot methods. The class attribute, a vector of character „class names“,

[20] which implicitly calls print()
[21] of the same column size for cbind, row size for rbind
[22] c() coerces all arguments to a common type. For a <- 1:3, b <- 4:5, compare c(a, b), cat(a, b), paste(a, b), and Cs(a, b) (in the Hmisc library), and further compare a, “a“, quote(a), as.character(a), print(a), dput(a), show(a), as.symbol(a), as.expression(a).
[23] This refers to the so-called S3 scheme. A new S4 scheme is available. S3 and S4 methods may exist side by side.

implements the *class* concept. The mere string becomes meaningful via *method dispatching*, i.e. with *generic functions* ramifying into class specific (or default) methods: A correct method is sought corresponding to the first element of the class attribute of the generic's first argument. If no such method is found, this class attribute's second element counts, and so on. If no method matches or if the generic's first argument has no class attribute, a default method is looked for. Important functions in the context of method dispatching are UseMethod() (to define a generic function) and NextMethod() (to implement inheritance); *group methods* may also be defined for the function groups ‚Math', ‚Summary' and ‚Ops'. unclass() removes an object's class; methods() informs about available methods/classes.

- Computing on the language:
  - Calls, expressions and functions are directly modifyable language objects. Calls have a list-like syntax; related functions are, e.g., quote(), eval(), as.call(), as.list(), as.name() or deparse().
  - substitute() replaces its first argument by another expression that may have passed through a function argument (useful: deparse(substitute(*arg*))) or is looked up in a list (e.g. substitute(*fun*(a), list(a=1))); this is related to *promises*. See also replace().
  - Objects of mode *expression* are of a similar structure as call objects; they are evaluated by eval().
  - Functions/closures may be directly manipulated by formals() (returning a (tagged) pairlist, in contrast to args()), body() and environment(), also on an assignment's left side. Note also the different effect of entering, e.g., q and q().
- *Global options* are stored in the list .Options, they are controlled by options(), getOption() and check.options().
- The operating system may be accessed through a number of functions, e.g. Sys.time(), file.copy() or dirname(); R is able to interface with foreign languages as C, e.g. through .C() or .External().
- Exception handling is controlled by functions as stop() or warning() (and warnings()), and the options warn, warning.expression and error. on.exit() provides function exit code (regardless of whether the exit was natural or not), e.g. for cleanup.
- Debugging is performed with the functions browser(), debug(), undebug(), trace() and untrace(). browser() and debug() provide a useful special prompt.

Appendix: Extensions and tables

- The XEmacs extension *ESS* provides an intelligent environment for R program editing. R's *Hmisc* library includes useful functions such as contents(), Cs(), describe(), label(), ldBands(), prn(), sas.get(), sedit(), src() and summarize().
- Basic vector types with associated mode and storage.mode:

type	mode	storage.mode
logical	*logical*	*logical*
integer	*numeric*	*integer*
double	*numeric*	*double*
complex	*complex*	*complex*
character	*character*	*character*

- Operators, in order of precedence (highest first; operators of equal rank are evaluated from left to right except where indicated):

[[[	indexing, binary
: :	name space/variable name separator
$ @	component/slot extraction, binary
^	exponentiation, binary (evaluated right to left)
- +	unary minus and plus
:	sequence operator, binary (in model formulae: interaction)
%xyz%	special binary operators; xyz may be replaced by anything. R already provides %% (modulus), %/% (integer divide), %*% (matrix product), %o% (outer product, equiv. to outer(arg1, arg2, "*")), %x% (Kronecker product), %in% (matching operator (in model formulae: nesting)).
* /	multiply, divide, binary
+ -	binary add, subtract
> < == >= <= !=	ordering and comparison, binary[24]
!	negation, unary
& &&	and (vectorized and non-vectorized), binary
| ||	or (vectorized and non-vectorized), binary
~	used in model formulae, unary or binary
-> ->>	assignment, binary
=	assignment (evaluated right to left), binary[25]
<- <<-	assignment (evaluated right to left), binary
?	help, unary or binary

- Probability distributions: The following distributions are available and may be used with the prefixes d (density, first argument x), p (CDF, first arg. q), q (quantile function, first argument p), and r (random deviate, first argument n[26]).

Distribution	R name	additional arguments[27]
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
Poisson	pois	lambda
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

[24] Caution with floating point numbers! E.g., try v <- seq(0.7, 0.8, by=0.1); v == 0.8. (Better: eps <- 1e-15; abs(v-0.8) < eps. Or even better: v <- seq(7, 8)/10; v == 0.8. The most accurate method is implemented in the function all.equal.numeric().)

[25] ‚<-' can be used anywhere, ‚=' is only allowed at the top level (i.e., in the complete expression typed by the user) or as one of the subexpressions in a braced list of expressions. In general, the use of ‚=' should be restricted to named function arguments. The „superassignment" operators ‚<<-' and ‚->>' or the function assign() may generally be used for global and permanent assignments within a function. As assign() does not dispatch assignment methods, it cannot set vector elements, names, attributes etc. Similarly, get() does not respect subscripting operators or assignment functions. (These two functions are not restricted to *names* that are *identifiers*.)

[26] nn for rhyper and rwilcox

[27] The ncp argument is currently almost only available for the CDFs. The p- and q-functions have the additional logical arguments lower.tail and log.p, the d-functions have log.