technische universität
dortmund

Dept. of Statistics

My first R package

Uwe Ligges

April 2011, Bordeaux, France

Contents

- Introduction and the usefulness of R packages
- Installation and administration of R packages in libraries
- Make the build tools work under Unix, Mac OS, and Windows
- Using R CMD build, INSTALL, check
- Development of R packages
    - Data
    - Functions
    - Documentation format and processing
    - C Code
    - Scoping issues
    - Namespaces
    - Debugging

Let me start with some excerpts of a beginners R course.

Benefits and drawbacks of R

Benefits

- Open Source
  - **Not a ‘black box’**
  - **Within current research**
  - **Extendability**
  - ...
- Support
  - ...

Drawbacks

- ...

What is R?

- A language and environment for data analysis and graphics
- Open Source
- **Tools for transfer of technology and methods using packages**
- Data access mechanism ...
- ...

Where can I get R from?

R has some homepage `http://www.R-Project.org` and there is the CRAN (Comprehensive **R** Archive Network):
`http://CRAN.R-Project.org`:

- **R** sources and binaries for some operating systems
- Almost 3000 **R packages** for various (statistical) methods
- ...

Functions

- All work is applied using **functions**.
- Defaults are documented on the **help pages**.
- ...
- *Everything* is an object (both data and functions)!

Help me!

Start the help system in a browser	`help.start()`
Help on a function	`help("functionname")` `?functionname`
Similar functions	`apropos("functionname")`
search by keyword	`help.search("keyword")`

Editors for R

In the **R** command line it is easy to quickly calculate things, but writing functions is not very convenient.
Hence it is recommended to choose an appropriate editor.

- A function can be saved in some kind of a text file on the hard disc and reloaded with `source("filename")`.
- Tiny functions and code pieces can be submitted via Copy&Paste.
- Syntax highlighting, auto-completion and other features are desirable.

Editors for R

- *ESS* (Emacs Speaks Statistics, `http://cran.r-project.org/other-software.html`) for the well known *Emacs* or *XEmacs* editor. With *ESS* it is possible to use *(X)Emacs* to control statistics software such as **R** and others conveniently.
- For Windows, the free editor *Tinn-R* (`https://sourceforge.net/projects/tinn-r`) is available
- as well as the *R–WinEdt* interface for the commercial editor *WinEdt* (not ready for WinEdt 6.x)

Packages

- **Package**: structured, standardized unit of **R** code, documentation, data, external code, ...
- Packages are loaded by `library("Packagename")` and unloaded by `detach()`.
- Help on packages (instead of functions) can be accessed by `library(help = "Packagename")`.
- On CRAN there are more almost 3000 packages available - on all (un)thinkable topics you can(not) imagine.
- The Omega(hat) and BioConductor projects are maintaining their own package repositories.
- An **R** standard installation loads the packages *base, datasets, graphics, grDevices, methods, stats* and *utils* on startup.
- Several package (including *base*) are shipped with R,
- as well as several important *recommended packages*.

additional ‘standard packages’

base	**R** base package
datasets	Collection of datasets
graphics	Graphics functions
grDevices	Graphics devices
grid	Re-design for graphics layout (e.g. for lattice)
methods	**S** 4 methods (Chambers, 1998)
splines	Splines
stats	Common statistical functions (tests, ...)
stats4	Same as stats with **S** 4 classes
tcltk	GUI programming with *tcl/tk*
tools	Tools for package development, administration, documentation
utils	Some helper functions

additional ‘recommended packages’

boot	Bootstrap methods (Davison and Hinkley, ’97)
cluster	Cluster methods (Rousseeuw et al.)
codetools	Code analysis
foreign	Import and export from and to Minitab, S, SAS, SPSS, Stata, .
KernSmooth	Kernel density estimation and smoothing (Wand & Jones, ’95)
lattice	Trellis graphics (Cleveland, ’93)
Matrix	Matrix classes (e.g. for sparse matrices)
mgcv	Generalized additive models
nlme	(Non-) linear models with mixed effects (Pinheiro & Bates, ’00
rpart	Recursive partitioning
survival	Survival analysis (hazard, Cox, censoring)

Packages by V&R

class	Classification
MASS	Collection of functions by Venables and Ripley (2002)
nnet	Neural nets (feed-forward) with one hidden layer — and multinomial log-linear models
spatial	Spatial statistics

Extensions

R is extremely extensible by the user. It is possible to

- write your own functions,
- generate standardized documentation for these functions,
- integrate *C*, *C++*, or *Fortran* code in form of a *shared library (DLL)*,
- create packages that include the before mentioned things and that can easily be installed and distributed.

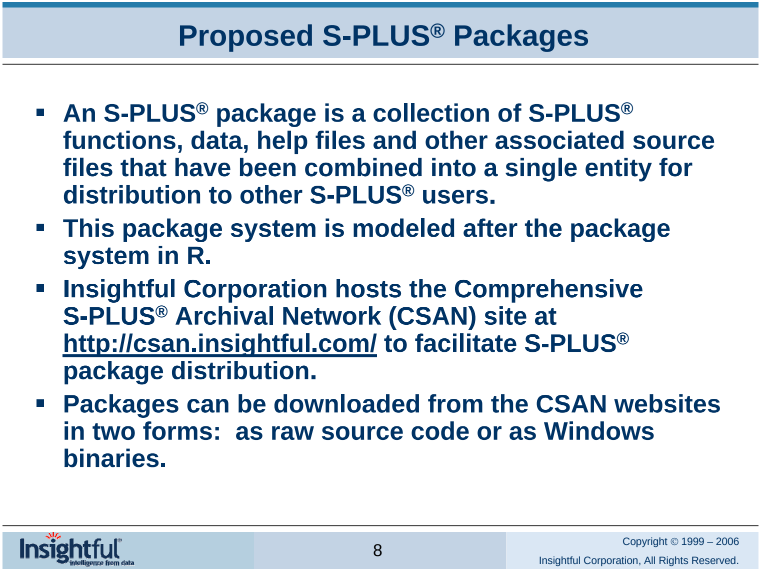
If you have written some useful code that implements some interesting method, you might want to publish it on *CRAN* in form of a package - like many others did already.

Why Packages?

Why should we package anything?

- Dynamical loading of packages (saves memory).
- Easy installation and update of packages (locally or from the web), within **R** or from the OS's command line.
- Easy administration – use global (department's server) and local libraries at the same time.
- Validation – **R** includes features for checking code, documentation and installability, as well as testing the results of pre-defined calculations.
- easy distribution to others using a standard mechanism.
- Example data.

The S-PLUS (8) package system and CSAN



Proposed S-PLUS® Packages

- An S-PLUS® package is a collection of S-PLUS® functions, data, help files and other associated source files that have been combined into a single entity for distribution to other S-PLUS® users.
- This package system is modeled after the package system in R.
- Insightful Corporation hosts the Comprehensive S-PLUS® Archival Network (CSAN) site at http://csan.insightful.com/ to facilitate S-PLUS® package distribution.
- Packages can be downloaded from the CSAN websites in two forms: as raw source code or as Windows binaries.

Insightful® intelligence from data 8 Copyright © 1999 – 2006 Insightful Corporation, All Rights Reserved.

Load packages from libraries

- Installed **R** packages live in a library, i.e. some directory
- and can be loaded from that library by
  `library("Packagename", lib.loc = Path_to_library)`
- `.libPaths()` shows which libraries are looked up for packages automatically
- A library can be added by `.libPaths()` to the search path
- or the library can be set before the start of **R** in the environment variable `R_LIBS`, e.g. in file `.Renviron`:
  `R_LIBS=/home/user/myR/myLibrary;/home/user/myR/develLibrary`
- Both base and recommended packages are in the main *library* in directory `R_HOME/library`
- `R_HOME` is the path that points to the current version of **R** , e.g. `/usr/local/lib/R` or `c:\Program Files\R-x.y.z`.
- Default is to install new packages into the first place of the result of `.libPaths()`.

Load packages from libraries

Examples:

```
library(help = "survival")  # help
library("survival")         # load
detach("package:survival")  # unload
.libPaths("c:/temp")        # set library
.libPaths()
```

Libraries

More than one library makes sense:

- Structuring packages
- Developer and user library
- central installation (no write permission for users) vs. local library of own packages

Examples:

- central library of standard packages, e.g. `n:\software\R-x.y.z\library`,
- central library of CRAN packages, e.g. `n:\software\Rlibs\CRAN`,
- central library of BioC packages, e.g. `n:\software\Rlibs\BioC`,
- local user library, e.g. `d:\something\myRlibs\work`,
- local developer library, e.g. `d:\something\myRlibs\devel`.

Package administration

Documentation:

- Manual ‘R Installation and Administration’
- ‘The R FAQ’ and ‘R for Windows FAQ’
- ‘R Help Desk: Package Management’ in *R News 3(3)*

Repositories:

- CRAN (+ CRAN extras for Windows), BioConductor, Omega
- `setRepositories()` or `options("repos" = ...)` for selecting repositories
- `chooseCRANmirror()` and `chooseBioCmirror()` for choosing mirror servers

Package administration

- `install.packages("package", lib = "/Path/to/library")`
  - automatically downloads the most recent version of a package from the repositories and installs it,
  - no need to specify `lib`, if the first place of the search path is the right library,
  - the argument `dependencies = TRUE` implies to install all declared dependent and suggested packages of the package.
- `update.packages()`
  - installs new versions of packages from the repositories
  - argument `checkBuilt = TRUE` implies recompiling of packages after a major upgrade of **R** .

Package administration

Summary of **R** functions:

available.packages()	packages in selected repositories
download.packages()	download packages
install.packages()	install packages
installed.packages()	locally installed package
new.packages()	package in repository that are not installed locally
old.packages()	locally installed package with newer versions in the repository
update.packages()	update package
contrib.url()	generates canonical form of repository
packageStatus()	considered to be the future (since several years)?!

Package administration – binary packages

The argument `type` in `install.packages()`, `update.packages()` and friends can be set to

- `"source"`
- `"win.binary"`
- `"mac.binary.leopard"`
- `"mac.binary"`

The default is the appropriate binary type on Windows and on the CRAN binary Mac OS X distribution, otherwise it is `"source"`. These can be overridden to install from sources under Windows, for example.

32- vs. 64-bit Windows binaries

Since R-2.12.0:

- use gcc 4.5.0 for 32-bit and gcc 4.5.2 for 64-bit R
- bi-arch binaries for both **R** and packages.

Package administration – binary packages

- Some tools are missing on typical Windows systems
- Windows shell (command line) differs from typical Unix systems
- For CRAN like repositories, R looks for packages in, e.g. `CRAN-mirror/bin/windows/contrib/2.12/`.
- `ReadMe` contains information what happened to packages not passing `R CMD check` .
- GUI available for **R** under Windows: „Packages“ provides the interface for `install.packages()` etc.
  (all installations into `.libPaths()[1]` !).

Package administration – local binary packages

Example:

Install the binary package `MyPackage` from the local file `c:\somewhere\MyPackage_0.0-1.zip` into `c:\myR\myLibrary`:

```
> install.packages(
+      "c:/somewhere/MyPackage_0.0-1.zip",
+      lib = "c:/somewhere/myLibrary", CRAN = NULL)
```

CRAN Task Views

- CRAN contains almost 3000 packages: Confusing!!!
- **CRAN Task Views**: Provide some summary and structure by topics
- grouping of packages (also by priority)
- administration package: ctv (Zeileis and Hornik, 2006)
- which structure is available: `available.views()`
- install all packages of one group: `install.views()`

Examples:

```
library("ctv")
(temp <- available.views())
temp[[8]]
install.views("MachineLearning", coreOnly = TRUE)
```

Source vs. binary packages

- **Source packages** are independent of the platform (hardware, operating system).
  - Prerequisites for installing source packages: Perl, C(++) compiler, Fortran compiler, ...
  - CRAN accepts only source packages
  - Standard way of distributing packages for Unix-like systems (Linux, Solaris, . . .).
- **Binary packages** are platform-specific and may depend on the **R** version in use.
  - Binary packages can be installed without prerequisites: ‘shared object files’ and DLL, help pages, meta information are already precompiled in a binary package.
  - CRAN provides binary packages for recent **R** versions for some platforms, e.g. Windows and MacOS X (PowerPC + Intel).
    Binary packages for Windows are provided roughly two days after the source packages appear.

Source vs. binary packages

- Distinction between binary and source packages by line starting with `Built:` in file `DESCRIPTION`:

```
Built: R 2.12.2; i386-pc-mingw32; 2011-04-11 09:30:00 UTC; w
```

- File extensions (by agreement):
  - `.tar.gz`: Source package
  - `.zip`: binary package for Windows
  - `.tgz`: binary package for Mac,
  - `.deb` or `.rpm`: binary package for Linux

Package administration II

For locally available source package, it is more common to use the OS's command line:

```
$ R CMD INSTALL -l /Path/to/library Paket
```

If `-l /Path/to/library` is not given (to specify the *library* explicitly):

- first *library* from environment variable `R_LIBS` is used
- main *library* is used
- `.Renviron` is not evaluated by `R CMD .....`

Source packages under Windows

Configure your environment:

- See: R Development Core Team (2011a), Ligges and Murdoch (2005)
- R tools: `http://www.murdoch-sutherland.com/Rtools`
  - collection of cygwin based shell tools
  - MinGW gcc (4.5.x) distribution
  - libraries for bitmap/jpeg support
  - vanilla perl
  - libraries for tcl/tk support
- LaTeX (e.g. MiKTeX): `http://www.miktex.org/`

Source packages under Windows

- Set paths (in environment variable 'PATH') to local (.) and all ...\bin paths (should happen automatically, if selected).
  ```
  PATH=.;c:\devel\tools\bin;c:\devel\MinGW\bin;
  c:\devel\R-2.12.2\bin;c:\devel\Perl\bin;
  c:\devel\texmf\miktex\bin;%PATH%
  ```
- Set environment variable 'TMPDIR' (otherwise 'TEMP' is used)

Structure of packages

A package consists of some standard files and directories, the latter containing certain files as described in the manual *Writing* **R** *Extensions*:

- `DESCRIPTION` (file) with standardized formatted entries for author, license, title, dependencies, ...
- `NAMESPACE` (file) for generating a Namespace
- `man/` (directory) contains documentation in `*.Rd` format.
- `R/` (directory) contains **R** code.
- `data/` (directory) contains data sets.
- `src/` (directory) contains *C*, *C++*, or *Fortran* sources.
- `tests/` (directory) contains files for validation.
- `demo/` (directory) contains **R** Code for demo purposes
- `inst/` (directory) contains stuff that is to be copied in the main directory of a binary package (e.g. Vignettes).

Except for the `DESCRIPTION` file all other items above are optional.

Package generation

Examples:

```
> package.skeleton(name = "MyPackage", ListOfObjects, path=".")
     Creating directories ...
     Creating DESCRIPTION ...
     Creating READMEs ...
     Saving functions and data ...
     Making help files ...
     Done.
     Further steps are described in ./MyPackage/README
```

Package generation

`package.skeleton()`:

- generates a skeleton for package `MyPackage`
- with files from `ListOfObjects`
- in the given path (here the current working directory)
- generates first version of the file `DESCRIPTION`
- generates first versions for the documentation file in `*.Rd` format – you just need to them fill out
- tells us what to do next

Next steps are:

- If all files have been edited, you can build the package by `R CMD build`.
- `R CMD INSTALL` installs the package.
- `R CMD check` checks for consistency, installability, documentation ...

Packages: Data and functions

- Each data set and each function lives in a separate file
    - regularly named by object name
    - function close to each other (such as generics with methods) are sometimes contained in one file
    - regularly with corresponding documentation in `/man`
- Data can be loaded with `data()` and has to be put into the `data/` directory in one of the formats:
    - ‘rectangular’ text file: separated by blank or comma, extension `.csv`, `.tab` or `.txt`
    - **R** source code written by `dump()` (extension `.r` or `.R`), and
    - **R** binary file written by `save()` (extension `.rda` or `.RData`).
- Code that should be executed once the package is loaded should go into the file `R/zzz.R`.

Packages: Documentation

- Help pages written in Rd format
- Manuals and reports: Package Vignettes with SWeave

Help pages:

- `package.skeleton()` prepares all Rd files for a package
- `prompt()` prepares a separate Rd file for one object to be documented
- $\LaTeX$ like syntax

Packages: Documentation

Example for an `*.Rd` file:

`\name`	Name of help page (commonly = `\alias`)
`\alias`	Name(s) of function(s) that are described
`\title`	title
`\description`	short description
`\usage`	function call including all arguments and their defaults
`\arguments`	description of all arguments and their meaning
`\value`	description of the returned value(s)
`\details`	more detailed description
`\references`	references (methods, implementation, algorithms)
`\seealso`	links to other relevant documentation of other functions
`\examples`	examples how to use the function
`\keyword`	standardized keyword

Packages: Documentation

- standardized defaults as well as self defined sections
- allow for mathematical formulas, URLs, links to other help pages, computation in and on help pages, etc.
- Layouted documentation from `*.Rd` files can be generated directly by
  - `R CMD Rdconv` for conversion to LaTeX, HTML and formatted ASCII text,
  - `R CMD Rd2dvi` for conversion to DVI and Adobe PDF.

Packages: Documentation

The **R** packaging system checks (using `R CMD check`) if:

- documentation is available for all (exported) data sets and functions in a package
- the `\usage` part corresponds to the actual definition of the function
- the code in section `\examples` can be executed without any error
- all the arguments of a function are documented
- all the defaults are documented
- `.Rd` files can be converted to the different formats

Vignettes

Vignettes

- are in the installed package in form of PDF files
- are in the source package in directory `./inst/doc`
- are shown with

```
vignette(package = "grid")
vignette("viewports", package = "grid")
```

SWeave

Generating vignettes using **SWeave** (Leisch, 2002):

- Code + Text:

  ```
  Text ...
  <<Options>>=
  Code chunk
  @
  ... more text.
  ```
- Sweave helps to integrate code and text automatically:
  - **R** evaluates the code and returns the results
  - LaTeX renders the text
- reproducible data analysis and research
- easily re-generate reports with minor changes in the data
- `R CMD check` checks whether code can be executed and evaluated
- there is something called `odfWeave` ...

Package, install and check a package

- **Package**, if all files have been generated:
  `R CMD build` builds the package and generates the vignettes
- **Install**: `R CMD INSTALL`
- **Check**: `R CMD check`
  - Consistency, installability
  - Documentation (as mentioned before)
  - Test cases (`.R` files) in directory `tests/`.
    Results (`.Rout` files) are compared with 'true' results (given as `.Rout.save` files)

R-forge

R-forge (`http://r-forge.r-project.org/`) is a **cental developer platform for R packages** offering easy access to the best in

- SVN
- daily built and checked packages
- mailing lists, message boards/forums
- bug tracking
- site hosting
- permanent file archival, full backups
- total web-based administration.

Submitting to CRAN

- Be sure your package passes the checks without any WARNINGs or ERRORs (in R-devel!).
- Upload the source (!) package to `ftp://cran.r-project.org/incoming`.
- Send e-mail message to `cran@r-project.org`.

What CRAN does

- Initial check of the package on Linux
- Make source package available in the repository
- Make binaries available for various OSs (within less than a week)
- Regular checks on different platforms
- Check summary pages: `http://cran.r-project.org/web/checks/check_summary.html`
- Package specific check summaries: `http://cran.r-project.org/web/checks/check_results_tuneR.html`
- Notifications in case the package is broken (by a change in a dependency or R itself)

Win-builder

- Builds Windows binaries and **checks for validation of the R base system**.
- Builds and checks new and updated packages – daily, at least for R-release and R-devel.
- Notification of developers.
- Daily build of R-devel.
- Re-check all packages for R-devel – weekly.
  Aim: Make new errors of packages or R itself quickly visible to developers.
- Public system to build and check your won packages under Windows if that is not available for you:
  `http://win-builder.r-project.org/`.

Win-builder

We need a check system that builds and checks **at least within 24 hours** for each flavor of R in order to

- provide check results when still of interest
- provide binaries directly after switching to alpha/beta/rc/release phase.

CRAN Windows Binaries' Package Check 2011

Last updated on 2011-04-04 09:50:06 **(Monday)** (simplified)

No	Package	Version	R-2.12.2	Inst. time	Check time
...	...	...	...	...	...
2953	ziccode	0.2	OK	3	27
2954	zipfR	0.6-5	OK	7	63
2955	zoeppritz	1.0-2	OK	1	16
2956	zoo	1.6-4	OK	4	69
2957	zyp	0.9-1	OK	2	18
Sum (in hours), **2x Xeon E5430 Quad:**				8.4/8	72.0/8

C, C++, or Fortran code

Why do we want to have compiled code?

- Speed
- Make use of already existing external efficient libraries

Calling compiled external sources can be done by the interfaces `.C()`, `.Call()`, `.Fortran()`, and `.External()`.

- A couple of important macros is defined in the header files `R.h` and `Rinternals.h`.
- Sometimes it is also useful to look into `Rdefines.h` for S4 and friends.

C, C++, or Fortran code

- Code is compiled automatically during package installation:
  `R CMD INSTALL` compiles code in the package (directory `src/`)
  - `dyn.load(filename)` loads and `dyn.unload()` unloads the resulting library
  - `library("packagename")` should load it, if in a package
  - `library.dynam()` can be used in function `.First.lib()` in `zzz.R`
  - or define it in your Namespace (later on)...
- `R CMD SHLIB` compiles the code without installing a whole package, i.e. you can invoke compiler and linker manually
- do never forget the garbage collector!

Example: C with .Call

As a simple **example** we are trying to add two real valued vectors *a* and *b* by a call through `.Call()`.

File `c:\test.c`:

```
#include <Rinternals.h>
SEXP add(SEXP a, SEXP b)
{
  int i, n;
  n = length(a);
  for(i = 0; i < n; i++)
    REAL(a)[i] += REAL(b)[i];
  return(a);
}
```

Example: C with .Call

- `add`, `a`, `b`: `SEXP` (*Symbolic EXPression*)
- returning the `a` – still an **R** object
- No new **R** object has been generated, hence no `PROTECT()` required

Example: C with .Call

Now we can generate a library from the C file `test.c` using `R CMD SHLIB` :

```
$ R CMD SHLIB test.c

gcc -I"t:/R/include" -O3 -Wall -std=gnu99 -c test.c -o test.o
gcc -shared -s -o test.dll tmp.def test.o -Lt:/R/bin -lR
```

Some files are generated now, particularly file `add.dll` (Windows) or `add.so` (Unix) respectively.

Example: C with .Call

R code:

```
dyn.load("c:/test.dll")          # load the library
# or library("Packagename"), if in some package ...

# Definition of the calling R function:
add <- function(a, b){
    if(!is.numeric(a) || !is.numeric(b))
        stop("a and b must be numeric")
    if(length(a) != length(b))
        stop("a and b must have same length")
    .Call("add", as.double(a), as.double(b))
}

add(4:3, 8:9)
```

Functions

- All work in **R** is done by functions.
- A function call has the form

  `functionname(argument1 = arg1, argument2 = arg2, etc.)`,

  where the arguments can be specified by name or not.
  - There are some special functions with convenient abbreviations such as `+`.

    You can rewrite `3 + 5` to its real function call: `"+"(3, 5)`.

    The name is not a regular one, hence the quotes.
  - An assignment has the full form: `"<-"(x, 3)`.
- There are arguments with defaults:
  - An argument without default *must* be specified in a function call.
  - An argument with default *may* be specified in a function call (and the default may be changed).

Functions

Write your own functions in order to collect a sequence of other function calls to do the same thing more than once, maybe with some parameters changed.

A function definition looks like this:

`MyFunction <- function(arguments){ statements }`,

where the `arguments` can be defined with or without defaults. When the function is called, the `arguments` are passed to the `statements`.

Statements may consist of several lines, as far as they are enclosed in braces (same is true for loops, for example).

Functions

A typical function definition might look like the following:

```
median <- function(x, na.rm = FALSE){
     # ... many lines of code! ...
    sort(x, partial = half)[half]
}
```

- There are two arguments: `x, na.rm`.
- Only the second argument has a default: `FALSE`.
- The last line of the function defines its value. More than one object can be returned as a list of objects. If `return()` is called, function evaluation stops and the argument of `return()` is returned.
- For a vector `a`, the following calls may be sensible:
  - `median(a)` (`na.rm` may be omitted, the default)
  - `median(a, TRUE)` (arguments ordered correctly, no names required)
  - `median(na.rm = TRUE, x = a)` (named arguments)

Functions

So we have to distinguish between *formal* arguments in a function's definition and *actual* arguments as specified in the function call. The rules to match actual and formal arguments are applied in the following way:

- At first, all arguments with completely given names are matched (`x = 1:10`).
- Then, arguments with partially given names are matched to the remaining *formal* arguments (`na = TRUE`).
- Next, all unnamed arguments are assigned in the given order to the remaining *formal* arguments.
- All remaining arguments are assigned to the three dots argument: `...`

You can test if a formal argument is missing in a call by `missing()`.

Functions

It is possible to use the formal 'three dots argument' `...` in the definition of a function. All non-matching *actual* arguments (in the sense of not matching to any other argument) are collected by `...`. This can be handled within the function or (what is more common) passed to other functions via `...`.

Examples:

```
ThreePoints <- function(x, ...){
    x <- x - 2
    median(x, ...)
}
x <- log(-1:100)
ThreePoints(x)
ThreePoints(x, na.rm = TRUE)
```

Lazy evaluation

R uses *lazy evaluation* of functions' arguments, i.e. statements used as actual arguments will be evaluated in their first usage, but not before:

Examples:

```
lazy <- function(x, calc = TRUE) {
    if(calc) x <- x+1
    print(a)
}
lazy((a <- 3), calc = FALSE)
lazy(a <- 3)

label <- function(x)
    return(list(call = substitute(x), value = x))
label(1+2)
```

Scoping rules

During programming, the question arises: ‘When are what objects visible for which functions?’

If you work in the **R** console directly, all new objects are created within the workspace.

In (more complex) functions many objects are generated that are only of temporary use. Hence it makes sense to evaluate functions in separate environments, in order not to clutter the workspace with unneeded objects. Therefore things are more transparent and less RAM is consumed.

This means assignments within a function will not be saved in the workspace. And objects from the workspace should be passed as arguments to functions that require those objects.

Scoping rules

Some more detailed comments related to *Scoping Rules* follow:

- **R** keeps all *environments* in its main memory (RAM)
- All top level generated **R** objects go into the workspace (‘.GlobalEnv’), number 0.
- There is some search path of environments containing packages (for functions) and data bases (for data.fram,es). At the center there is the ‘.GlobalEnv’ (workspace), at the end the base package and in between some objects added to the path by calls to `library()` or `attach()`.
- If a function is called, a new *environment* (starting with number 1) is created.
- If a function is called within the former function, the next environment is generated.

Scoping rules

- Search rule is that a function looks for objects (a) in its own environment, (b) the one of its parents, (c) the workspace and (d) all the attached packages and data bases.
- If a function returns, its environment is deleted (incl. all the objects it contains). Therefore you have to `return()` objects for further use.
- The functions `assign()` and `get()` can assign objects to or get objects from arbitrary *environments*.

Scoping rules

```
-8 package:base
-7 Autoloads
... ... ... ... ...
-2 package:methods
-1 package:stats
 0 .GlobalEnv        # Workspace
 1 environment 1     # Function 1
 2 environment 2     # Function 2
 3 environment 3     # Function 3
```

Type `search()` for the current search path.

Scoping rules

Examples:

```
scope <- function()
{
    x <- 3
    inner <- function()
        print(x)
    inner()
}

scope()     # --> R: 3  # --> S-Plus: ERROR
x <- 5
scope()     # --> R: 3  # --> S-Plus: 5
```

Scoping rules

R is capable of so called *Lexical Scoping* (Gentleman, R. and Ihaka, R., 2000).

This means a function that has been created in some specific *environment* and assigned to some object outside of the function afterwards, always knows all object of the originating *environment*. Therefore, under such circumstances, an *environment* is not deleted (but only if no function has been returned).

This feature might be beneficial but also confusing (because scoping rules are different). In the latter case also consult Venables, W.N. and Ripley, B.D. (2000).

There are some more exceptions from the described scoping rules, most important one is implemented by namespace rules which will be described later.

Scoping rules

Examples:

```
l.scope <- function()
{
    only.here <- 2
    newFoo <- function()
        print(only.here)
    return(newFoo)
}
value <- l.scope()

value()   # --> R: 2  # --> S-Plus: ERROR
only.here <- 4
value()   # --> R: 2  # --> S-Plus: 4
```

Namespaces

Some more rules (in addition to the known scoping rules, how to search objects in existing environments) have been introduced by R's *Namespaces* support.

- The number of contributed packages increases almost daily, hence you can expect name clashes of function between all those packages.
- Namespaces define which objects are visible to the user and to other functions, and which are only visible within the own namespace.
- Functions that are not *exported*, are only visible within the own namespace (and hidden to the user).
- A namespace's objects are independent of names of other namespaces' functions.

Namespaces

Consider you define

```
foo <- function(x) sin(2 * pi * x)
```

then you probably expect that the objects `sin()` and `pi` are from package **base**. If there are functions with the same names in other packages or the workspace, the latter objects would be found before those in base:

```
foo <- function(x)
    sin(2 * pi * x)
foo(1:5)         # Expected: [1] -2.449213e-16 -4.898425e-16
sin <- sum
pi <- 0.5
foo(1:5)         # Sum of (1:5) = 15
```

Namespaces

- A namespace guarantees that no objects from **base** are masked for functions in other namespaces.
- You can explicitly *import* objects from other namespaces. These cannot be accidently overloaded afterwards. Packages loaded by *import* directives are not attached to the search path.
- A function from some namespace looks for objects according to the following rules: at first it looks into the own namespace, then into imported objects or namespaces, then into the **base** namespaces, and then the already known scoping rules are applied.

Namespaces

- For explicit access to an object in a package with namespace the '`::`' operator can be used, which separates the name of the namespace and the object's name. Hence, `stats::ks.test` accesses the object (function) `ks.test` in namespace `stats`.
- In rare cases, you want to access non exported functions which can happen by calling `getFromNamespace()`.
- The operator '`:::`' can access a non exported object as well.
- `fixInNamespace()`: change / replace a function within a namespace.
- `getS3method()`: access a non-exported method.
- `getAnywhere()`: all objects in the search path and loaded namespaces are looked up.

Namespaces

Examples:

```
library("MASS")                    # load MASS
lda                                # function lda: generic
methods(lda)                       # Which methods?
lda.default                        # lda.default is not exported
getS3method("lda", "default")      # look at it anyway ...
getAnywhere("lda.default")
MASS:::lda.default
```

the file NAMESPACE

The file `NAMESPACE` in the toplevel directory of your package:

- define objects to be imported and exported:
  `export()` and `exportPattern()` (for exporting many objects at a time)
- define code to be loaded (in form of an external library such as a `DLL`):
  `useDynLib()`
- define S3 methods:
  `S3method()`
- `import()` imports a whole namespace, `importFrom()` imports objects from another namespace
- S4 objects:
  - `exportClasses()`, `exportMethods()`
  - `importClassesFrom()`, `importMethodsFrom()`

the file NAMESPACE

Example:

```
useDynLib(myPackage)
export(foo2)
S3method(print, myClass)
import(klaR)
importFrom(MASS, lda)
```

Debugging

- If you write your own functions, you will make mistakes!
- If it is a small function, it may be easy to find the error.
- In more complicated functions it may be worse to find a bug, leading to nervous breakdowns.
- **R** offers some tools for easy debugging.
- It is advisable to debug your own package with deactivated Namespace (i.e. just rename the `NAMESPACE` file and reinstall), otherwise see `?debugInNamespace`.

Beside those tools, you can print (`print()`, `cat()`) objects or informative texts to the console, of course.

Debugging with tools

- `traceback()` shows which function has caused the last error, including the stack ('path') of calls. This way you can find the bad function even within very encapsulated function calls.
- `debug(foo)` enables debugging for the function `foo`, i.e. it will be executed within some *browser* (see below; until debugging is turned off again with `undebug(foo)`).
- `browser()` starts the *browser* at this place within a function.
- `recover()` and `options(error = recover)`: If an error emerges, the *browser* is started so that you can jump into one of the environments that existed at the time where the error occured.

Debugging with tools

Examples:

```
foo1 <- function(x){  |foo1 <- function(x){   |foo1 <- function(x){
 foo2 <- function(x,s)| foo2 <- function(x,s){| foo2 <- function(x,s){
   x[[s]] + 5         |   browser()           |   print(x)
 y <- x + 1           |   x[[s]] + 5          |   x[[s]] + 5
 foo2(y, s = -5)      | }                     | }
}                     | y <- x + 1            | y <- x + 1
                      | foo2(y, s = -5)       | foo2(y, s = -5)
                      |}                      |}
                      |                       |
foo1(1:5)             |foo1(1:5)              |foo1(1:5)
traceback()           |                       |options(error = recover)
                      |                       |foo1(1:5)
```

References — Core manuals

Online at `http://CRAN.R-Project.org/manuals.html` and in R:

- R Development Core Team (2011a): *R Installation and Administration*. ISBN 3-900051-09-7.
- R Development Core Team (2011b): *R Language Definition*. ISBN 3-900051-13-5.
- R Development Core Team (2011c): *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0.
- R Development Core Team (2011d): *Writing R Extensions*. ISBN 3-900051-11-9.

The R Journal (formerly *R News*): `http://journal.r-project.org/`.

References — R I

- Chambers, J.M. (2008): *Software for Data Analysis: Programming with R*, Springer, New York.
- Gentleman, R. and Ihaka, R. (2000): Lexical Scope and Statistical Computing. *Journal of Computational and Graphical Statistics 9*, 491–508.
- Ihaka, R. and Gentleman, R. (1996): **R**: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics 5*, 299–314.
- Leisch, F. (2002): *Sweave User Manual*. `http://www.ci.tuwien.ac.at/~leisch/Sweave`
- Ligges, U. (2003): R Help Desk: Package Management. *R News 3(3)*, 37–39.
- Ligges, U. and Murdoch, D. (2005): R Help Desk: Make ‘R CMD’ Work under Windows - an Example. *R News 5(2)*, 27–28.

References — R II

- Murdoch, D. and Urbanek, S.(2009): The New R Help System. *The R Journal 1(2)*, 60–65.
- Ripley, B.D. (2004): Lazy loading and packages in R 2.0.0. *R News 4(2)*, 2–4.
- Ripley, B.D. (2005a): Internationalization features of R 2.1.0. *R News 5(1)*, 2–7.
- Ripley, B.D. (2005b): Packages and their management in R 2.1.0. *R News 5(1)*, 8–11.
- Venables, W.N. and Ripley, B.D. (2000): *S Programming*, Springer, New York.
- Venables, W.N. and Ripley, B.D. (2002): *Modern Applied Statistics with S*, 4th ed., Springer, New York.
- Zeileis, A. and Hornik, K. (2006): *ctv: CRAN Task Views*. R package version 0.3-2.