Advanced R Programming

Duncan Murdoch

Department of Statistical and Actuarial Sciences
University of Western Ontario

June 10, 2007

Outline

Part I

Introduction to R

Outline

Outline

First there was S...

S began as a project at Bell Laboratories in 1976, involving John Chambers, Rick Becker, Doug Dunn, Paul Tukey, and Graham Wilkinson.

> *We wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important.*

— John Chambers, in Stages in the Evolution of S

The New S Language

R was based on version 3 of S (released around 1988). Allan Wilks, Trevor Hastie and many others had joined the effort.

> *It had become a functional, object-based language, in its own fashion. Everything is now an object, including functions and expressions.*
>
> *. . .*
>
> *The main impetus was to present a simple user interface to dealing with the diverse kinds of data encountered in fitting and examining statistical models.*

— John Chambers, in Stages in the Evolution of S

The Birth of R

Around 1992, Ross Ihaka and Robert Gentleman at the University of Auckland needed statistical software for a teaching lab. S-PLUS didn't run on the Macintosh; they decided to write their own. Ross was very impressed with Scheme (a Lisp dialect); they based the initial code on the design of Scheme interpreters.

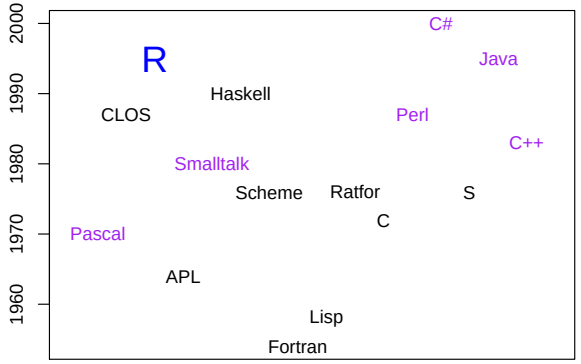
> *To make the interpreter useful, we had to add data structures to support statistical work and to choose a user interface. We wanted a command driven interface and, since we were both very familiar with S, it seemed natural to use an S-like syntax.*

— Ross Ihaka, in R: Past and Future History

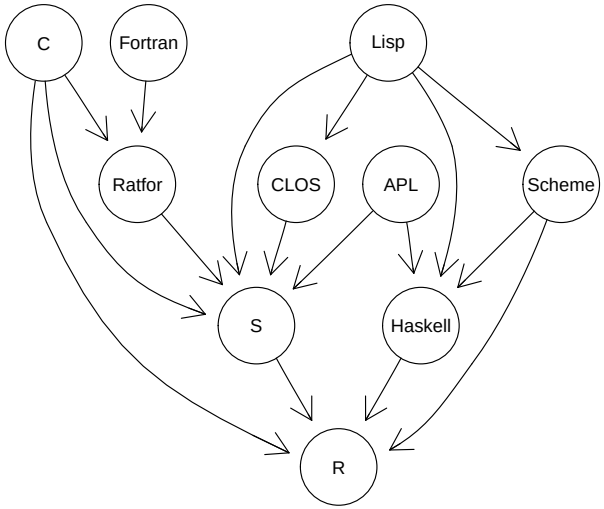
GNU S

- The R's announced R on the `s-news` mailing list in 1993; Martin Maechler convinced them to release it as an open source project under the GPL, which they did in 1995.
- In 1997 the R Core group was established, consisting of Doug Bates, Peter Dalgaard, Robert Gentleman, Kurt Hornik, Ross Ihaka, Friedrich Leisch, Thomas Lumley, Martin Maechler, Paul Murrell, Heiner Schwarte, and Luke Tierney.
- Since then John Chambers, Stefano Iacus, Guido Masarotto, Duncan Murdoch, Martyn Plummer, Brian Ripley, Duncan Temple Lang and Simon Urbanek have joined R Core; Guido Masarotto and Heiner Schwarte have left.

Where does R fit in?



R Language Influences



R is Open Source S

- S is like APL in showing the user a workspace with a calculator interface.
- S syntax is C-like in the definitions of binary operators, but it is more like Fortran in indexing arrays. Control structures are C-like, with some idiosyncracies:

```
> for (name in vector) {}
```

- R syntax and S syntax are almost identical. (R has imported some C-like ways of specifying constants, e.g. 0x10 is hexadecimal for 16, 1L indicates that the value is to be stored as an integer.)

But there are differences...

- R imported the idea of lexical scope from Scheme and Haskell:

```
> adder <- function(x) {
+   return( function(y) x + y )
+
+ }
> f <- adder(10)
> f(5)
[1] 15
```

- R has its own tree-like search list.

Outline

The World According to R

- The user has a workspace of R objects (the “global environment”).
- R has its own objects, in a search list of environments.
- R is a computing engine, waiting for commands from the user to create new objects and manipulate old ones.
- There’s also a file system on the host computer, and possibly some graphics devices and other external things like databases, etc.

Simple Computations

```
> ls()
character(0)
> x <- 1
> ls()
[1] "x"
> x
[1] 1
> search()
[1] ".GlobalEnv"        "package:stats"
[3] "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"
[9] "package:base"
```

(Almost) Everything is a Vector

Most R objects are vectors, and most operations work element-by-element.

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> y <- 6:10
> y
[1]  6  7  8  9 10
> x + y
[1]  7  9 11 13 15
```


Types of Objects

Every object in R has a type. The vector types are:

- The atomic vectors: `logical`, `integer`, `double`, `complex`, `character` and `raw`.
- List vectors: `list`

Other types include:

- `closure` (interpreted function)
- `environment`
- `NULL`
- Other types: `symbol`, `pairlist`, `promise`, `language`, `special`, `builtin`, `char`, `expression`, `externalptr`, `weakref`, `S4`, plus a few others.

Examples

```
> typeof(TRUE)
[1] "logical"
> typeof("hello")
[1] "character"
> typeof(1 + 2i)
[1] "complex"
```


Examples

How a value prints doesn't always indicate its type:

```
> as.integer(1)
[1] 1
> typeof(as.integer(1))
[1] "integer"
> 1
[1] 1
> typeof(1)
[1] "double"
```

Examples

In R, the `typeof()` function is the main way for a program to determine the type. There are also `mode()` and `storage.mode()`, mainly for S compatibility.
A user friendly function is `str()` (for "structure"):

```
> str(x)
 int [1:5] 1 2 3 4 5
> str(1)
 num 1
> str(as.integer(1))
 int 1
```


Examples

```
> typeof(list(3, "abc", 1.3))
[1] "list"
> list(3, "abc", 1.3)
[[1]]
[1] 3

[[2]]
[1] "abc"

[[3]]
[1] 1.3
```


Recursive Ttypes versus Atomic Types

- Lists are a *recursive* type. Each element of a list can be of a different type.

```
> x <- list(3, "abc", 1.3)
> typeof(x)
[1] "list"
> typeof(x[[2]])
[1] "character"
```

- Atomic vectors must be all of one type.

```
> y <- c("a", "b", "c")
> typeof(y)
[1] "character"
> typeof(y[2])
[1] "character"
```


Outline

Several Different Systems

R has two underlying graphics models:

- Classic S graphics is an "ink on paper" model: functions add things to the current graphics device, possibly covering what was there, but only erasing the whole frame at once.
- Paul Murrell's `grid` graphics is based on a hierarchical model of a scene, with transformations possible at each level. It is much more flexible than classic graphics, allowing objects to be changed, but is not interactive.

High Level Graphics

- Classic high level graphics are supported by the `graph` package.
- Deepayan Sarkar's `lattice` package implements a version of Cleveland's trellis graphics using `grid`.
- Hadley Wickham's `ggplot` package implements a version of Leland Wilkinson's *Grammar of Graphics* using `grid`.

Other Systems

- `rggobi` is an interface to the GGobi package for interactive graphics.
- `rgl` is a package that provides rotatable 3-D graphics, with some classic S-like plotting functions.
- `scatterplot3d` does static 3-D graphics within the classic graphics engine.

Learning about Graphics

We're not going to teach graphics today, but I highly recommend Paul Murrell's book *R Graphics* for those who want to learn either classic S graphics or `grid`.

Outline

Obtaining R

R is downloadable from

`http://cran.r-project.org`

or mirrors (e.g. `http://cran.ca.r-project.org`). Lately the US mirror has been way behind on updates; avoid it.
There are *binary* and *source* versions available. The binary versions are compiled for a particular platform. For Windows,

`http://cran.r-project.org/bin/windows/base/R-2.5.0-win32.exe`

A `.dmg` file is available for MacOSX.
The source version is common to all platforms, and requires compiling before it can be used.

Installing R

The Windows and MacOSX binary versions include standard installers for those platforms. Just open them and follow the instructions.
I recommend installing to a folder with no spaces in the name (e.g. `C:\R`) to avoid confusing some of the Unix tools used later.

Part II

Advanced Programming

Outline

Outline

(Almost) Everything is an Object

- One of the great strengths of the S language is that almost everything is an object.
- Even *functions* are objects.
- You can do computations on functions: “computing on the language”.

Writing Functions

Functions have 3 parts: the header, the body, and the environment. For example,

```
> mymean <- function(x) {
+     n <- length(x)
+     sum <- sum(x)
+     return(sum / n)
+ }
```

- The header is `function(x)`.
- The body is the block of three lines in braces.
- The environment determines the context in which the lines are evaluated.

Function Headers

- The header of the function determines how it looks from the outside.
- It is formed as `function(`*`list of arguments`*`)`.
- The arguments are one of
  1. *name*, a required parameter.
  2. *name = value*, an optional parameter with a default value.
  3. ..., an optional list of unspecified parameters.

Parameter Matching

- If you give names, partial matching may be used.
- Unnamed parameters are matched by position.
- Don't confuse yourself with code like this! —

```
> f <- function(firstarg, secondarg) {
+     cat("firstarg=", firstarg,
+         "secondarg=",secondarg,"\n")
+ }
> f(1,first=2)
firstarg= 2 secondarg= 1
```


The Function Body

- The body of a function is the statement following the header. Usually we put several statements in braces for a complex function, but the braces are not necessary.
- When you invoke (call) the function, an *evaluation frame* is created, containing objects corresponding to each argument in the header.
- Assignments in the body create objects in this evaluation frame.
- The value of the function is the last value calculated, or (better!) the value passed to `return()`.

Lazy Evaluation

Formal arguments to the function are created as *promises*, which are not evaluated until used.

```
> f <- function() cat("f was called\n")
> g <- function() cat("g was called\n")
> h <- function(x, y) y
> h(f(), g())
g was called
```

Promises

- You can see the expression associated with a promise using `substitute()`.

```
> showarg <- function(x) {
+     cat("The expression was",
+         deparse(substitute(x)), "\n")
+     cat("Its value is", x, "\n")
+ }
> showarg( 1 + 1 )
The expression was 1 + 1
Its value is 2
```

- Promises may also be created explicitly, using `delayedAssign()`.
- For historical reasons `substitute()` doesn't work properly in the global environment.

Environments

- The evaluation frame is an example of an *environment*, a type of R object similar to a list, but:
- They are unordered; objects in them can only be retrieved by the exact name.
- Assignment only creates a new reference to the same environment, not a new copy.
- They have a *parent* environment; (some) searches for objects will continue the search in the parent if the object is not found in the current environment.

Environments cont'd

```
> x <- 123
> e <- new.env()
> e$y <- 456
> e$x
NULL
> e$y
[1] 456
> get("x", e)
[1] 123
> get("y", e)
[1] 456
```


Evaluation

Expressions are always evaluated in an environment:

- The global environment when typed at the console.
- The evaluation frame when evaluated in a function body.
- Wherever you want:

```
> y <- 321
> evalq(x + y)
[1] 444
> evalq(x + y, e)
[1] 579
```

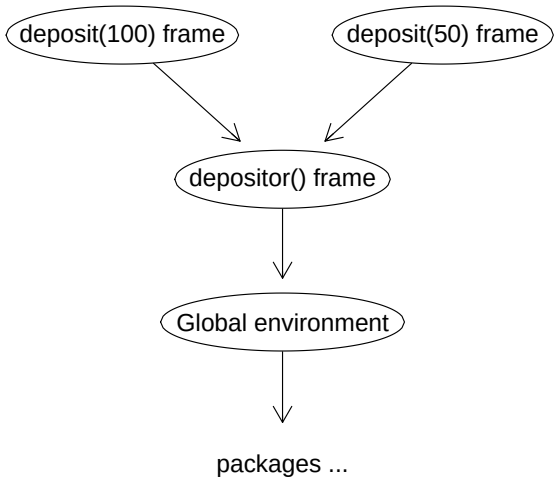
The Function Environment

- A function's environment is normally the environment in which the function was created:
  - the global environment for most user functions
  - the evaluation frame for functions created in other functions
  - the namespace of a package
- The parent of the evaluation frame is the function environment.

Environment Example

```
> depositor <- function() {
+     balance <- 0
+     function(amount) {
+         balance <<- balance + amount
+         return(balance) }
+ }
> deposit <- depositor()
> environment(deposit)$balance
[1] 0
> deposit(100)
[1] 100
> deposit(50)
[1] 150
```


Environment Example cont'd



Default Arguments

- Arguments passed to the function are evaluated in the environment of the caller.
- Default values for skipped arguments are evaluated in the evaluation frame.

```
> rect <- function(height = width/2,
+                  width = height*2) {
+   list(height=height, width=width)
+ }
> rect(height = 4)
$height
[1] 4

$width
[1] 8
```

Outline

Many Different Paradigms!

R has inherited 3 different ideas of "objects" from S:

1. Everything is an object of one of the fixed types described earlier.
2. R supports S version 3 objects: a fairly unstructured scheme based on the `class` attribute.
3. R supports S version 4 objects: a highly structured scheme based on CLOS.

And that's not all...

There are other schemes in packages as well:

- The `R.oo` package by Henrik Bengtsson supports a style of object somewhat similar to Delphi, Java or C++.
- The `tcltk` package provides an interface to a TCL/TK interpreter for GUI objects.
- The `RJava` package provides an interface to a Java interpreter.
- Other experimental packages...

S Version 3 Objects

This is the easiest and least sophisticated style of object-oriented programming.

- Objects have a `class` attribute (a character vector of class names).
- *Generic functions* (e.g. `print`) can automatically call *methods* based on the class of the first argument.

```
> print
function (x, ...)
UseMethod("print")
<environment: namespace:base>
> head(methods(print))
[1] "print.acf"     "print.anova"
[3] "print.aov"     "print.aovlist"
[5] "print.ar"      "print.Arima"
```

S Version 3 Example

```
> covariate <- rnorm(100)
> response <- 1.5 + 3.1*covariate + rnorm(100)
> fit <- lm(response ~ covariate)
> typeof(fit)
[1] "list"
> class(fit)
[1] "lm"
> fit
Call:
lm(formula = response ~ covariate)

Coefficients:
(Intercept)    covariate
      1.533        3.278
```


Common S3 generics

It is standard practice to supply methods for at least some of the following generics:

- `print()`, `summary()`, `plot()` for display
- `formula()`, `predict()` and `residuals()` for model-fitting functions
- `model.frame()`, `model.matrix()`, `case.names()`, `variable.names()` for programming with regression-like models.

What's Wrong with S3?

Some problems:

- Is `t.test()` the `t()` method for class `test`? (No.)
- What if several arguments have classes? (Only the first one matters.)
- Anyone can define new generics; how can I be sure my class will work with them? (I can't.)

S Version 4 Objects

A new object system was introduced by John Chambers in his *Programming with Data* book. This system is based on CLOS:

- Formal definitions: `setGeneric()`, `setMethod()`, etc.
- Function dispatch is based on the *signature* of the call to the generic: the classes of *all* parameters.
- Multiple inheritance is formalized in several ways.

No S4 Today...

We won't be studying S4 methods today:

- It is a very rich system; it would take a whole day by itself!
- I find it too rich for my taste: it is hard to design, hard to debug.
- I think it still has the S3 problem that anyone can define new generics for which my class will fail, but this is open to debate.

Outline

Getting it Right

Debugging has 5 steps:

1. Recognize that a bug exists.
2. Make the bug reproducible.
3. Identify the cause of the bug.
4. Fix the error and test.
5. Look for similar errors.

R provides help with the first three steps.

Recognizing a Bug

- Validate the input to functions using `stop()` and `stopifnot()`, and provide warning messages:

```
> mymean <- function(x) {
+     stopifnot(is.numeric(x))
+     if (any(is.na(x)))
+         warn("input contains NA")
+     return( sum(x)/length(x) )
+ }
```

Recognizing a Bug cont'd

- Put together test cases, and test them! (More on this in package building).
- Worry when a bug "just goes away": track it down!
- Don't ignore warnings.

Making the Bug Reproducible

- Use `source()` to read scripts and execute them: then you know what was typed.
- Use `set.seed()` to set the random number seed to a known value, so that random errors are reproducible.

Identifying the Cause of a Bug

- `cat()` and `print()` provide old-fashioned ways to "instrument" your code: just print things as you go.
- The `browser()` function can be called anywhere in your functions. It allows you to print and modify objects in the evaluation frame (or elsewhere).
- The `debug()` function marks a function for debugging. The next time you enter that function, R will halt execution and enter the browser.
- `options(error = recover)` enters the browser when an error occurs.
- `options(warn = 2)` converts warnings to errors.
- Mark Bravington's `debug` package provides a GUI for debugging.

Getting it Fast Enough

Code optimization has 3 steps:

1. Get it right.
2. Get it fast enough.
3. Make sure it's still right.

Profiling helps with step 2: when your code isn't fast enough, you need to know which parts are the bottlenecks, and which parts don't matter.

The R Profiler

R includes a profiler that works by interrupting execution at regular time intervals (e.g. 50 times per second), and recording the call stack at the time of the interruption. (The call stack records which functions are currently being evaluated, which functions called them, and so on.)

Profiling Example

Which is faster, `mean()` or `var()`?

```
> mean
function (x, ...)
UseMethod("mean")
<environment: namespace:base>
```


Profiling Example cont'd

```
> head(mean.default,10)
1  function (x, trim = 0, na.rm = FALSE, ...)
2  {
3      if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
4          warning("argument is not numeric or logical: returning NA
5          return(NA_real_)
6      }
7      if (na.rm)
8          x <- x[!is.na(x)]
9      if (!is.numeric(trim) || length(trim) != 1)
10         stop("'trim' must be numeric of length one")
```


Profiling Example cont'd

```
> tail(mean.default,-10)
11     n <- length(x)
12     if (trim > 0 && n > 0) {
13         if (is.complex(x))
14             stop("trimmed means are not defined for complex da
15         if (trim >= 0.5)
16             return(stats::median(x, na.rm = FALSE))
17         lo <- floor(n * trim) + 1
18         hi <- n + 1 - lo
19         x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
20     }
21     .Internal(mean(x))
22 }
```

Profiling Example cont'd

```
> var
function (x, y = NULL, na.rm = FALSE, use)
{
    if (missing(use))
        use <- if (na.rm)
            "complete.obs"
        else "all.obs"
    na.method <- pmatch(use, c("all.obs", "complete.obs", "pairwise
    if (is.data.frame(x))
        x <- as.matrix(x)
    else stopifnot(is.atomic(x))
    if (is.data.frame(y))
        y <- as.matrix(y)
    else stopifnot(is.atomic(y))
    .Internal(cov(x, y, na.method, FALSE))
}
<environment: namespace:stats>
```


Profiling Example cont'd

```
> set.seed(123)
> Rprof()
> junk <- replicate(100000, {
+     x <- rnorm(1000)
+     mean(x)
+     var(x)
+ })
> Rprof(NULL)
```


Profiling Example cont'd

```
> head(summaryRprof()$by.self, 18)
```

	self.time	self.pct	total.time	total.pct
rnorm	30.30	66.7	30.30	66.7
var	3.18	7.0	11.32	24.9
stopifnot	2.12	4.7	5.86	12.9
mean.default	2.00	4.4	2.06	4.5
pmatch	1.44	3.2	1.52	3.3
mean	1.20	2.6	3.26	7.2
eval	0.98	2.2	1.20	2.6
match.call	0.74	1.6	1.06	2.3
inherits	0.62	1.4	0.62	1.4
any	0.50	1.1	0.58	1.3
==	0.30	0.7	0.30	0.7
FUN	0.28	0.6	45.16	99.5
all	0.26	0.6	0.26	0.6
lapply	0.18	0.4	45.34	99.9
sys.parent	0.18	0.4	0.18	0.4
is.data.frame	0.14	0.3	0.76	1.7
sys.call	0.14	0.3	0.32	0.7
!	0.14	0.3	0.14	0.3

Profiling Example cont'd

```
> head(summaryRprof()$by.total, 18)
```

	total.time	total.pct	self.time	self.pct
replicate	45.40	100.0	0.00	0.0
sapply	45.40	100.0	0.00	0.0
lapply	45.34	99.9	0.18	0.4
FUN	45.16	99.5	0.28	0.6
rnorm	30.30	66.7	30.30	66.7
var	11.32	24.9	3.18	7.0
stopifnot	5.86	12.9	2.12	4.7
mean	3.26	7.2	1.20	2.6
mean.default	2.06	4.5	2.00	4.4
pmatch	1.52	3.3	1.44	3.2
eval	1.20	2.6	0.98	2.2
match.call	1.06	2.3	0.74	1.6
is.data.frame	0.76	1.7	0.14	0.3
inherits	0.62	1.4	0.62	1.4
any	0.58	1.3	0.50	1.1
sys.call	0.32	0.7	0.14	0.3
==	0.30	0.7	0.30	0.7
all	0.26	0.6	0.26	0.6

Part III

Add-on Packages

Outline

Outline

Why Write Packages?

Why write packages?

- Packages are a great way to disseminate statistical methods or datasets.
- Packages are a great way to organize your own code.

Why are they so great?

- Packages have a standard format that makes sense.
- There are tools to check packages for common errors.
- There are tools to install packages on multiple platforms.

What are Packages?

- Packages are collections of files organized in a specified directory structure.
- Packages may be collected into an archive file (`.zip`, `.tar.gz` or `.tgz`, for example).
- *Source* packages are what you write; *binary* packages have been compiled and are ready to use.
- `R CMD build` converts source code directories to source packages in `.tar.gz` format.
- `R CMD INSTALL –build` or `R CMD build –binary` compiles them into binary format (`.zip` on Windows, `.tgz` on MacOSX).

Standard Structure

The *Writing R Extensions* manual gives the details on what is needed in a package. Standard parts:

DESCRIPTION A structured file containing information about the package: title, description, authors, dependencies, etc.

R/ A subdirectory containing R code.

man/ A subdirectory containing documentation.

The `package.skeleton()` function can produce skeleton versions of the required subdirectories and files.

Optional Subdirectories

data/ Datasets stored as R code to generate them, as tables readable by `read.table`, or as saved images.

src/ C, C++, FORTRAN 77, Fortran 9x, Objective C and/or Objective C++ source code and header files. There may also be Makevars or Makefile, but these are not usually necessary.

demo/ R code to be run by `demo()`.

inst/ Any sort of file to be installed by R: they are moved up one directory.

tests/ R scripts to run tests, and optionally saved output from successful runs of the tests.

exec/ Other executables that the package may need, e.g. shell scripts.

Optional Files

NAMESPACE A file describing the imports and exports from the package.

INDEX A rarely used file describing each object included in the package.

configure, cleanup Scripts used in building the package.

COPYING A description of the license for your package.

Other files Other files (such as README, NEWS or ChangeLog) which will be ignored by R. Put them in inst/ if you want them installed.

The NAMESPACE File

The NAMESPACE file describes what your package exports and what it imports from other packages. It is optional, but I *strongly* recommend using one.

- You can feel free to have functions in your package for your own use, without exporting and documenting them.
- You can document where your package will look for the functions it uses, and avoid clashes if a user happens to have a function of the same name as a standard one.

NAMESPACE Structure

Here is part of the NAMESPACE file for the `rgl` package:

```
export(asEuclidean, asHomogeneous, aspect3d,
       axes3d, axis3d, box3d, bbox3d, bg3d,
       clear3d, cube3d, decorate3d, dot3d, ... )

S3method(dot3d, qmesh3d)
S3method(wire3d, qmesh3d)
S3method(shade3d, qmesh3d)
...
if(tools:::.OStype() == "windows") {
  import(utils) # only needed for getWindowsHandle
}
```


Documentation Files

There are two main formats for R documentation:

1. .Rd files in the man/ subdirectory, which use a type of markup somewhat like LaTeX. See the *Writing R Extensions* manual for details.
2. Vignettes in Sweave format in the inst/doc/ subdirectory. Sweave format is a mixture of R code interspersed with LaTeX(or some other text). For details, see the Sweave man page or the Sweave manual at `http://www.ci.tuwien.ac.at/~leisch/Sweave`.

Building, Installing, and Checking

- Once you have written a package, you may want to collect all the files into a source package archive for distribution: Use the `R CMD build packagedir` command in the shell for this.
- If your package builds without errors, then you should install it to make it available to use. The shell command for this is `R CMD INSTALL package.tar.gz`.
- You should use `R CMD check package.tar.gz` to run formal tests of your package. This will point out missing documentation, inconsistencies, etc.

Outline

Outline