Creating Packages in R
In House Presentation

Joshua Wiley

Statistical Consulting Group
University of California, Los Angeles
wileyj@ucla.edu

Wednesday, 10 August, 2011

Oh, gosh, this is getting whimsical. R, I surmise, is a mutable tool that enables the user to do what they want. Knapped flint for the Stone Age statistical fool, plasma arc for the expert and learned savant. R is a friend to all manner of men. The perfect companion, the servant complete. It gently informeth the clueless and then, reveals the essence unto the esthete.

– Jim Lemon (in a discussion about the usability of R)
R-help (May 2006)

Table of Contents

Background

- R is composed of a core set of packages that provide infrastructure and basic functionality. These are well maintained and thoroughly tested.
- Minor versions are released every 6 months
- Over 3,200 community developed add on packages (available from CRAN) extend functionality to virtually any task

Please stop me if you have questions!

Technical Notes I

- The R is based on S and is (mostly) a functional language
- Core R only runs on a single thread presently[1]. This is in the process of changing.
- **R runs in memory**[2] — operations on data are typically fast as long as the data "fit"
- R is case sensitive
- Many functions are vectorized (operate on entire vector at once)
- There are a number of GUIs and other programs coming out to make getting started easier (e.g., Rstudio & Deducer)[3]

Getting Started I

Let's look at a simple example session.

```
> x <- c(1, 3, 5, 7, 9)
> x

[1] 1 3 5 7 9
```

I **c**ombined the numbers into a vector assigned using "<-" to `x`. Functions are called by their name, followed by an open and close parenthesis "()". Arguments are separated by a ",". Here, the numbers were the arguments. When I typed `x` at the console, `R` **show**ed the values of the vector back to me. Note the number on the left—R indexes starting at **1** not 0. Because I did this assignment at the console, `x` is in the global environment[4].

Technical Notes II

You just saw the most common type of data in R — a vector of class numeric. Vectors are the basic data structure that underlies everything else. *A vector can only contain one class of data*.

- Class hierarchy—the highest class determines class of vector
- raw $<$ logical $<$ integer $<$ real $<$ complex $<$ character $<$ list
- matrices and arrays *are* vectors with more complex indexing
- each element of a list can be a vector $\therefore$ each element of a list can contain different data classes
- data frames *are* lists with each vector constrained to equal length
- data frames in R parallel data sets in SPSS, SAS, etc.

Getting Started II

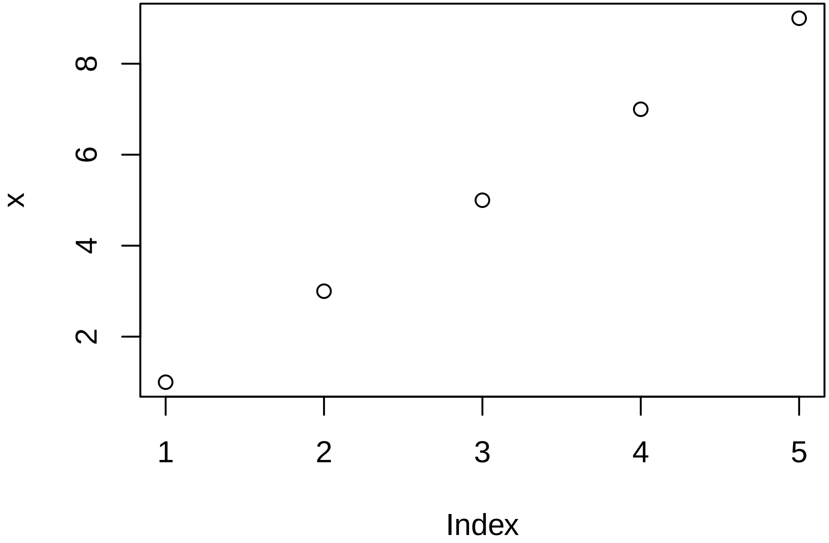
These are basic functions that work on many types of data structures and classes.

```
> str(x)

 num [1:5] 1 3 5 7 9

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
      1       3       5       5       7       9 
```

First we saw the `structure` of "x" and then got a simple summary.

Getting Started III

```
> plot(x)
```



Technical Notes III

Missing values are represented as `NA` and do not behave as other data.

```
> x <- c(x, NA)
> x

[1]  1  3  5  7  9 NA

> x > 3

[1] FALSE FALSE  TRUE  TRUE  TRUE    NA

> is.na(x)

[1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

Many statistical functions use the argument `na.rm = TRUE` to **re**move missing values (often not the default).

Getting Started IV

We saw c combine two vectors. Let's look how "x" is treated with an NA value.

```
> length(x)

[1] 6

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
      1       3       5       5       7       9       1 

> mean(x)

[1] NA

> mean(x, na.rm = TRUE)

[1] 5
```

R usually forces the user to choose how to handle missingness.

Technical Notes IV

`as.*` functions attempt to coerce class

```
> class(x)

[1] "numeric"

> as.character(x)

[1] "1" "3" "5" "7" "9" NA

> x <- c(x, "4")
> class(x)

[1] "character"

> as.numeric(x)

[1]  1  3  5  7  9 NA  4
```

always know your data str()ucture before you manipulate or analyze it.

Table of Contents

Introduction

In the previous section, we saw the basics of how R works with and handles data. This section is devoted to how to find out information on your own. What do you do if you are lost and on your own? We will cover

- How to learn about specific functions
- How to learn how to use a function
- How to find a function that does what you want

What does `foo` do?

Bring up the relevant documentation/help page by typing one of:

```
> help("mean")
> ?mean
```

Sections of R Documentation

R documentation files have a standard format.

Title	The title of the function(s), data, or whatever being documented
Description	A few sentence summary of what the purpose of the topic is
Usage	An example of the function written out (though not with real data). How it would be used.
Arguments	All the possible arguments to the function(s) including types, valid values, names, etc.
Details	Further details often to augment the arguments section or explain “how”
Value	What is returned by the function?
Examples	Ready-to-use examples of the function at work

Demo here

Using functions I

Figuring out how to use a new function can be tricky[5]. These tools help.

```
> ## show all the argument names
> args(help.search)

function (pattern, fields = c("alias", "concept", "title"), ap
    keyword, whatis, ignore.case = TRUE, package = NULL, lib.l
    help.db = getOption("help.db"), verbose = getOption("verbo
    rebuild = FALSE, agrep = NULL, use_UTF8 = FALSE)
NULL

> ## have examples run at the console
> example(mean)
```

Using functions II

```
mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50

mean> mean(USArrests, trim = 0.2)
  Murder  Assault UrbanPop     Rape
    7.42   167.60    66.20    20.16
```

Note the change in the prompt when examples are run.

What if I do not know foo?

```
> apropos("cov")
[1] "ability.cov" "cov"           "cov.wt"      "cov2cor"
[5] "covratio"    "discoveries" "recover"     "vcov"
> ## use the 'sos' package
> install.packages("sos")
package 'sos' successfully unpacked and MD5 sums checked

The downloaded packages are in
        C:\Users\Joshua Wiley\AppData\Local\Temp\RtmpoxAafz\do
> require(sos)
> ## not run
> # findFn("multinomial logistic")
```

Also look at crantastic.

Demo here

Table of Contents

Introduction

In the previous section, we saw the basics of how R works with and handles data. In this section, we will look at how to work with more realistic data structures. We will cover

- Matrices, data frames (the real workhorse), and lists
- Extraction — how to access specific pieces of your data
- Glimpse at how to use functions on whole datasets or many variables at once

Matrices I

Matrices are simply vectors that are indexed on two dimensions. They are indexed with typical `[row, column]`.

```
> X <- matrix(1:10, nrow = 2)
> X
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

As suggested by the row and column headings, particular cells can be extracted using '[' (the extraction operator)

```
> X[1, 4]
```

```
[1] 7
```

Now lets compare extraction with the vector x with the matrix $\mathbf{X}$

Matrices II

```
> x[1]

[1] 0

> X[1, 1]

[1] 1
```

But remember I said matrices *are* vectors, so we can *also* index them as vectors

```
> X[1:5] # extract first 5 elements of the matrix

[1] 1 2 3 4 5
```

Notice the numbers are 1, 2, 3, 4, 5. R indexes by columns not rows.

Matrices III

Now that we are dealing with more complex structures, the class will be different:

```
> class(X) # X is a matrix
[1] "matrix"
> str(X) # 2 x 5 integer matrix
 int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
```

data frames I

On the surface, data frames seem very similar to matrices. They are also two dimensional matrix-like objects. The important difference is that each column of a data frame is its own vector. This allows for different classes in each column (“variable”).

```
> dat <- data.frame(Age = c(50, 30, 40, 50),
+   Name = c("Jill", "Jack", "Jennifer", "John"))
> dat
```

```
  Age     Name
1  50     Jill
2  30     Jack
3  40 Jennifer
4  50     John
```

data frames II

Notice that essentially I created two vectors with `c` and passed both of them to the `data.frame` function. Adding `Age =` named the variables[6].

```
> class(dat)

[1] "data.frame"

> str(dat)

'data.frame':         4 obs. of  2 variables:
 $ Age : num  50 30 40 50
 $ Name: Factor w/ 4 levels "Jack","Jennifer",..: 3 1 2 4
```

data frames III

`str()` demonstrates something very important. The “Name” variable is not a character. It was automatically converted to a factor. The factor class is special in that underneath it is numeric, but has character label representations. When used models, some form of contrast codes will be automatically applied to factor class variables[7].

What happens if we try to convert the data frame to a matrix?

```
> as.matrix(dat)
```

```
     Age  Name
[1,] "50" "Jill"
[2,] "30" "Jack"
[3,] "40" "Jennifer"
[4,] "50" "John"
```

data frames IV

Note that everything is character class now (indicated by the double quotes). Matrices only contain one class and character is higher than numeric, so everything is elevated to character.

data frames V

Data frames can be indexed like a matrix

```
> dat[2, 1]

[1] 30
```

Or using the variable names

```
> dat[2, "Age"]

[1] 30

> dat[3:4, "Age"] # rows 3 AND 4

[1] 40 50
```

Earlier I said that data frames *are* lists with the added constraint that each element has to be the same length (so it works as a table). This means there are some more options for extraction.

data frames VI

```
> dat$Age

[1] 50 30 40 50

> dat[["Age"]]

[1] 50 30 40 50

> dat["Age"]

  Age
1  50
2  30
3  40
4  50
```

data frames VII

The $ is a convenient shortcut and often used. Note that it is *not* quoted. You also cannot easily pass variables to it[8]. Also compare the double vs. single bracket. Double brackets drop the name of the variable and just returns the vector, single brackets keep the data frame form.
If one of the dimensions is left blank (for both matrices and data frames), the entire dimension is selected.

```
> dat[1, ] # all of row one

  Age Name
1  50 Jill

> X[2, ] # all of row two

[1]  2  4  6  8 10
```

lists I

Lists are useful because they can be used to combine almost any set of objects. Each element of a list can be a vector (as in data frames), or the element could itself be a data frame, or matrix, or the results of a model fit you want to store. Thus lists are extremely flexible ways to say, “this set of objects belongs together although it may not be as structured as a table”.

```
> mylist <- list(1, 2, 3)
> mylist

[[1]]
[1] 1


[[2]]
[1] 2
```

lists II

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

Here you see a simple list with three elements, each a vector of length 1. It is unnamed, so it can only be extracted using:

```
> mylist[[1]]

[1] 1

> mylist[1]

[[1]]
[1] 1
```

lists III

A slightly more complex example

```
> mylist <- list(A = 1, B = 2, C = 3)
> mylist

$A
[1] 1


$B
[1] 2


$C
[1] 3

> mylist$A

[1] 1
```

lists IV

```
> mylist[[1]] #can still use the numbers

[1] 1

> mylist[["A"]]

[1] 1

> mylist["A"]

$A
[1] 1
```

Here you can see different ways of extracting from a named list.

lists V

Finally we will look at a complex, nested list structure.

```
> mylist <- list(df = dat, mat = X, vec = x,
+   char = c("ABCDEF..."))
> mylist

$df
  Age     Name
1  50     Jill
2  30     Jack
3  40 Jennifer
4  50     John


$mat
     [,1] [,2] [,3] [,4] [,5]
```

lists VI

```
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

$vec
 [1]  0  1  2  3  4  5  6  7  8  9 10 50

$char
[1] "ABCDEF..."
```

Here you can see that “mylist” contains in the first element the data frame we made, then the matrix, the vector, and another short character vector. To better understand the types of data in the list:

```
> class(mylist)

[1] "list"
```

lists VII

```
> str(mylist)
```

```
List of 4
 $ df  :'data.frame':        4 obs. of  2 variables:
  ..$ Age : num [1:4] 50 30 40 50
  ..$ Name: Factor w/ 4 levels "Jack","Jennifer",..: 3 1 2 4
 $ mat : int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
 $ vec : num [1:12] 0 1 2 3 4 5 6 7 8 9 ...
 $ char: chr "ABCDEF..."
```

There are special functions that apply (or use) a function *on each element* of a list. Here is a simple example find the `dimensions` of each object stored in the list.

```
> lapply(X = mylist, FUN = dim)
```

lists VIII

```
$df
[1] 4 2

$mat
[1] 2 5

$vec
NULL

$char
NULL
```

lists IX

This may seem a rather silly example, but it can become quite powerful allowing, for example, easy ways to extract the same coefficient from numerous model objects. In a simulation study, one could store each model in a separate list element and then in one line of code extract all of the residuals, or all the parameter estimates, or extract and then average the covariance matrix.

Table of Contents

Introduction

Packages in R provide a mechanism for organizing and sharing code and data. Various features (such as documentation and built in checks) assist in making trustworthy code.

Packages are not difficult to create, but there are many little steps and details that must be followed. Alhough not necessarily required, there are a variety of tools beyond R that are desirable to have if one is developing packages.

For users of the Windows OS, a tool set is provided that bundles appropriate versions of all of these tools. For *nix users, many of these tools are included by default (e.g., C compilers, tar, etc.)

In these slides, I will briefly demonstrate some of the steps and talk conceptually. More detailed, prescriptive steps are available.

Making I

Before developing your own package, I would try installing R and other package from source. This will let you work out any issues related to having the necessary software available and knowing how to use it.
On Windows, I would first download and install the toolset. Then download the sources (these are provided as tarballs). From there open the shell and

```
> tar -xf R-Major.minor.patch.tar.gz
> cd \directory\where\tar\ball\was\unpacked
REM tweak some install settings as needed
> cd \to\where\the\make\files\are
> make all recommended
```

Making II

That (roughly) gets you R compiled from source. You will need to make sure the /bin/ directory for the version of R you want to be using is added to the Windows `PATH` environment variable (this can be tricky if you do not have admin privileges).
Now you can install any package tar balls easily.

```
> R CMD INSTALL /path/Package_name.version.tar.gz
```

and the package is installed. Suppose you have the raw source (from your own package, for instance) and you want to make a tar ball, just add the build option. A variety of other options exist and can be seen as usual in a shell `R CMD INSTALL --help`.

```
> R CMD INSTALL --build /path/Package_name.version.tar.gz
```

Preparing for your Package

After you have a few functions written, it may be time to put them together in a package. A number of utility functions help partially automate the process of creating the appropriate directories and files. Specifically, `package.skeleton` can be used with several code files or functions to create a package skeleton. Once this initial step has been done, to continue adding later, use `prompt`, `promptMethods`, `promptClass`.
See `?package.skeleton` and the Writing R Extensions manual for more details.

A Simple Package I

The most basic package requires a description file and a namespace file, and subdirectories for the R code ("R"), the documention ("man"), and data (if present, "data").
The description file describes the package. This is used by `R` to determine things like the package version, other packages your package depends on, etc.

Demo here

This is created automatically by `package.skeleton`, but you have to fill it in. Most the fields make sense, but note the "Depends", "Imports", and "Suggests" lines. These list packages that my package *depends* on (`R`, the core packages, `methods`, and some graphcis ones);

A Simple Package II

imports (i.e., uses select functions from, but does not necessarily require them to be fully loaded to operate); and suggests (may enhance functionality or be similar, but not needed).
R knows about certain licenses, so for GPL-2 and GPL-3, it is enough to specify their name. GPL-3 is the preferred now, I believe. Briefly, it makes provisions that your code may be copied, changed, and redistributed, but if it is still your code it must be redistributed freely, open source, and able to be altered and reredistributed, etc.

The Namespace I

The development version of `R` requires that all packages have a namespace. If there is not one, it tries to create one, but you should write your own.
How R uses namespaces is non-trivial and has to do with lexical scoping[1].
In the package namespace file, you just need to specify the functions (or classes or methods) that you want to be available to users.
For instance, I might write some high level function `WriteManuscript()` that I want users to be able to directly call to write their manuscripts for them. `WriteManuscript` in turn calls some internal (non exported) workhorse functions `RARecruit()`, `ReviewerRebuttler()`, `Dejargonfier()`.
These "workhorse" functions are not meant to be used directly. Reasons may include requiring very specialized input, not checking for valid values (that may be assumed to be done by the higher level `WriteManuscript`).

The Namespace II

If needed, nonexported functions can be accessed using fully qualified references. For example

```
> require(Jmisc) # load my package
> cat(try(star)) # not found because not expored
Error in try(star) : object 'star' not found
```

The Namespace III

```
> Jmisc:::star # fully qualified reference works

function (x)
{
    symnum(x, legend = FALSE, na = FALSE, cutpoints = c(0, 0.0
        0.01, 0.05, 1), symbols = c("***", "**", "*", ""))
}
<bytecode: 0x00000000053cb6c8>
<environment: namespace:Jmisc>
```

The form is PackageName:::FunctionName.

The namespace file also specifies which functions should be imported from other packages. "Depends" will be fully loaded and attached, "Imports" will be loaded but not attached to the search path.

Demo namespace file here

[1]drawings and explanations here if desired

R Code

Adding R code to packages is probably the simplest aspect. Simply create the text files named with the *.R extension. You can create separate files for each function, or bundle multiple functions in one file. It will be processed identically. All the pure R code files go into the /R/ directory.

Demo directory and file

R Documentation I

Trickier than including code is creating appropriate documentation for it. This is where the `prompt*` functions we learned about earlier will help. They create general stubs that meet the requirements. Part of the built in checks look to make sure that every function has a corresponding documentation file and that every argument in the function's code matches the documentation of the arguments.

Table of Contents

Notes

Most of the resources mentioned in this presentation are freely available on the WWW. Hyperlinks are included for all of these as they are mentioned in the text. These were active as of August 10, 2011.

Endnotes I

[1] There are add on packages, notably snow, snowfall, Rmpi, multicore, foreach, doMC

[2] Workarounds exist but tend to be problem specific (e.g., use a data base, incremental computations for linear models, etc.)

[3] I am old fashioned and just prefer a solid text editor (Emacs with Emacs Speaks Statistics)

[4] Every package has its own special environment, and the insides of functions have environments. User's typically do not need to care about this, but it can become important when you have the same variable name used in different environments with different data.

[5] Conscientious package writers make sure their functions are clearly named, thoroughly documented, and come with examples of common usage.

[6] Note it is also possible to name columns and rows in matrices, but it is less common because different columns do not necessarily represent different variables.

[7] so-called "treatment" contrasts by default for unordered factors and orthogonal polynomial contrasts for ordered factors

[8] so if you have all the columns you want to extract stored in a character vector, you would need to use '[' not '$'