

Programming for Chemical and Life Science Informatics

I573 - Week 7
(Statistical Programming with R)

Rajarshi Guha

24th February, 2009

Resources

- Download [binaries](#)
 - ▶ If you're working on Unix it's a good idea to compile from [source](#)
- There's lots of documentation of varying quality
 - ▶ The R [manual](#) is a good document to start with
 - ▶ [R Data Import and Export](#) is useful when you need to work data saved in other packages
 - ▶ A variety of short introductory tutorials as well as documents on specific topics in statistical modeling can be found [here](#)
- More general help can be obtained from
 - ▶ The [R-help](#) mailing list. Has a lot of renowned experts on the list and if you can learn quite a bit of statistics in addition to getting R help just by reading the archives of the list. Note: they do expect that you have done your homework! See the [posting guide](#)
 - ▶ Some [reference tables](#) that map commands from Numpy and Matlab to R
 - ▶ A useful [reference](#) for Python/Matlab users learning R

- A number of books are available ranging from introductions to R along with examples up to advanced statistical modeling using R.
- [Data Analysis and Graphics Using R: An Example-based Approach](#) by John Maindonald, John Braun
- [Introductory Statistics with R](#) by Peter Dalgaard
- [Using R for Introductory Statistics](#) by John Verzani
- If you end up programming a lot in R, you'll want to have [Programming with Data: A Guide to the S Language](#) by John M. Chambers

- You can start learning R at the prompt
- Anything more than 3 lines, will be easier done in an editor
- The simplest workflow is to keep R open and an editor open and cut and paste from the editor to the R window
- As always, life becomes easier with a proper editor
 - ▶ Syntax coloring
 - ▶ Brace matching
 - ▶ Integration with R

IDE's and Editors

- Emacs + [ESS](#) - works on Linux, Windows, OS X. *Invaluable* if you are an Emacs user
- [TINN-R](#) is a small and useful editor for Windows
- You can also do R in Eclipse
 - ▶ [StatET](#)
 - ▶ [Rsubmit](#)
- A number of different GUI's are available for, which may be useful if you're an infrequent user
 - ▶ [Rcommander](#)
 - ▶ [JGR](#)
 - ▶ [Rattle](#)
 - ▶ [PMG](#)
 - ▶ [SciViews-R](#)

What is R?

- R is an environment for modeling
 - ▶ Contains many prepackaged statistical and mathematical functions
 - ▶ No need to implement anything
- R is a matrix programming language that is good for statistical computing
 - ▶ Full fledged, interpreted language
 - ▶ Well integrated with statistical functionality
 - ▶ More details later
- Used in a wide variety of areas
 - ▶ Chemometrics, cheminformatics, bioinformatics
 - ▶ Ecology, econometrics, finance, geograph, political science

What is R?

- Similar to Matlab, since both are vector/matrix oriented environments
- Matlab is better suited for numerical simulation etc., whereas R is not so good for this
- R is excellent for statistical modeling, though Matlab also has extensive support
- R is open-source, free and well maintained. No need to worry about licenses
- R is developed and supported by some of the top statisticians and many cutting edge techniques get implemented in R

What is R?

- It is possible to use R just for modeling
- Avoids programming, preferably use a GUI
 - ▶ Load data → build model → plot data
- But you can also get much more creative
 - ▶ Scripts to process multiple data files
 - ▶ Ensemble modeling using different types of models
 - ▶ Implement brand new algorithms
- R is good for prototyping algorithms
 - ▶ Interpreted, so immediate results
 - ▶ Good support for vectorization
 - ★ Faster than explicit loops
 - ★ Analogous to `map` in Python and Lisp
 - ▶ Most times, interpreted R is fine, but you can easily integrate C code

What is R?

- R integrates with other languages
 - ▶ C code can be linked to R and C can also call R functions
 - ▶ Java code can be called from R and vice versa. See various packages at rosuda.org
 - ▶ Python can be used in R and vice versa using [Rpy](#)
- R has excellent support for publication quality graphics
- See [R Graph Gallery](#) for an idea of the graphing capabilities
- But graphing in R does have a learning curve
- A variety of graphs can be generated
 - ▶ 2D plots - scatter, bar, pie, box, violin, parallel coordinate
 - ▶ 3D plots - OpenGL support is available

Working in R

- To get help for a function name do: `?function_name`
- If you don't know the name of the function, use:
`help.search('blah')`
- [R Site Search](#) and [Rseek](#) are very helpful online resources
- To exit from the prompt do: `quit()`
- Two ways to run R code
 - ▶ Type or paste it at the prompt
 - ▶ Execute code from a source file: `source('mycode.R')`
- Saving your work
 - ▶ `save.image(file='mywork.Rda')` saves the whole workspace to *mywork.Rda*, which is a binary file
 - ▶ When you restart R, do: `load('mywork.Rda')` will restore your workspace
 - ▶ You can save individual (or multiple) objects using `save`

Basic Types in R

Primitive Types

- **numeric** - indicates an integer or floating point number
- **character** - an alphanumeric symbol (string)
- **logical** - boolean value. Possible values are TRUE and FALSE

Complex Types

- **vector** - a 1D collection of objects of the same type
- **list** - a 1D container that can contain arbitrary objects. Each element can have a name associated with it
- **matrix** - a 2D data structure containing objects of the same type
- **data.frame** - a generalization of the matrix type. Each column can be of a different type

Primitive Types

```
> x <- 1.2
```

```
> x <- 2
```

```
> y <- 'abcdefg'
```

```
> y
```

```
[1] "abcdefg"
```

```
> z <- c(1,2,3,4,5)
```

```
> z
```

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

```
> z <- 1:5
```

```
> z
```

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

Matrices

- Make a matrix from a vector
- The matrix is constructed column-wise

```
> z <- c(1,2,3,4,5,6,7,8,9,10)
> m <- matrix(z, nrow=2, ncol=5)
> m
```

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

data.frame

- We want to store compound names and number of atoms and whether they are toxic or not
- Need to store: characters, numbers and boolean values

```
> x <- c('aspirin', 'potassium cyanide', 'penicillin', 'sodium hydroxide')
> y <- c(21, 3, 41, 3)
> z <- c(FALSE, TRUE, FALSE, TRUE)
> d <- data.frame(x,y,z)
> d
```

	x	y	z
1	aspirin	21	FALSE
2	potassium cyanide	3	TRUE
3	penicillin	41	FALSE
4	sodium hydroxide	3	TRUE

- But do the column means? 6 months later, we probably won't know (easily)
- So we should add names

```
> names(d) <- c('Name', 'NumAtom', 'Toxic')
> d
```

	Name	NumAtom	Toxic
1	aspirin	21	FALSE
2	potassium cyanide	3	TRUE
3	penicillin	41	FALSE
4	sodium hydroxide	3	TRUE

- When adding column names, don't use the underscore character
- You can access a column of a data.frame by it's name: `d$Toxic`

- A 1D collection of arbitrary objects (ArrayList in Java)

```
> x <- 1.0
> y <- 'hello there'
> z <- TRUE
> s <- c(1,2,3)
> mylist <- list(x,y,z,s)
> mylist
[[1]]
[1] 1.0

[[2]]
[1] "hello there"

[[3]]
[1] TRUE

[[4]]
[1] 1 2 3
```

- We can access the lists by indexing: `mylist[[1]]` for the first element or `mylist[c(1,2,3)]` for the first 3 elements

- It's useful to name individual elements of a list

```
> x <- 'oxygen'  
> z <- 8  
> m <- 32  
> mylist <- list(name='oxygen', atomicNumber=z, molWeight=m)  
> mylist  
$name  
[1] "oxygen"  
  
$atomicNumber  
[1] 8  
  
$molWeight  
[1] 32
```

- We can then get the molecular weight by writing

```
> mylist$molWeight  
[1] 32
```

Identifying Types

- It's useful initially, to identify the type of the object
- Usually just printing it out explains what it is
- You can be more concise by doing

```
> x <- 1
> class(x)
[1] "numeric"

> x <- 'hello world'
> class(x)
[1] "character"

> x <- matrix(c(1,2,3,4), nrow=2)
> class(x)
[1] "matrix"
```

Indexing

- Indexing fundamental to using R and is applicable to vectors, matrices, data.frame's and lists
- all indices start from 1 (not 0!)
- For a 1D vector, we get the i 'th element by `x[i]`
- For a 2D matrix of data.frame we get the i, j element by `x[i, j]`
- But we can also index using vectors
 - ▶ Called an *index vector*
 - ▶ Allows us to select or exclude multiple elements simultaneously

Indexing

- Say we have a vector, x and a matrix y

```
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

- Some ways to get elements from x
- 5th element $\rightarrow x[5]$
- The 3rd, 4th and 5th elements $\rightarrow x[c(3,4,5)]$
- Everything **but** the 3rd, 4th and 5th elements $\rightarrow x[-c(3,4,5)]$

Indexing

- Say we have a matrix `y`

```
> y
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

- Some ways to get elements from `y`
- Element in the first row, second column \rightarrow `y[1,2]`
- All elements in the 1st column \rightarrow `y[,1]`
- All elements in the 3rd row \rightarrow `y[3,]`
- Elements in the first 2 columns \rightarrow `y[, c(1,2)]`
- Elements **not** in the first 2 columns \rightarrow `y[, -c(1,2)]`

Indexing

- You can also use a boolean vector to index another vector
- In this case, the index vector must of the same length as your target vector
- If the i 'th element in the index vector is TRUE then the i 'th element of the target is selected

```
> x <- c(1,2,3,4)
> idx <- c(TRUE, FALSE, FALSE, TRUE)
> x[ idx ]
[1] 1 4
```

- It's a pain to have to create a whole index vector by hand

Indexing

- A more intuitive way to do this, is to make use of R's vector operations
- `x == 2` will compare each element of `x` with the value 2 and return TRUE or FALSE
- The result is a boolean vector

```
> x <- c(1,2,3,4)
> x == 2
[1] FALSE TRUE FALSE FALSE
```

- So we can select the elements of `x` that are equal to 2 by doing

```
> x <- c(1,2,3,4)
> x[ x == 2 ]
[1] 2
```

Indexing

- Or select elements of `x` that are greater than 2

```
> x <- c(1,2,3,4)
> x[ x > 2 ]
[1] 3 4
```

- Or select elements of `x` that are 2 *or* 4

```
> x <- c(1,2,3,4)
> x[ x == 2 | x == 4 ]
[1] 2 4
```

- The single bar (`|`) is intentional
- Logical operations using vectors should use `|`, `&` rather than the usual `||`, `&&`

Functions in R

- Functions are much like in any other language
- Arguments are always passed by value
- Typing is dynamic

```
my.add <- function(x,y) {  
  return(x+y)  
}
```

- You can then call the function as `my.add(1,2)`
- If you send in the wrong types the function will report an error
- In general, check for the type of object using `class()`

Loop Constructs

- R does not have C-style looping

```
for (i = 0; i < 10; i++) {  
  print(i)  
}
```

- Rather, it works like Python's for-in idiom

```
for (i in c(0,2,3,4,5,6,7,8,9)) {  
  print(i)  
}
```

Loop Constructs

- In general, to loop n times, you create a vector with n elements, say by writing $1 : n$
- But like Python loops, you can just loop over elements of a vector or list and use them

```
> objects <- list(x=1, y='hello world', z=c(1,2,3))  
> for (i in objects) print(i)  
[1] 1  
[1] "hello world"  
[1] 1 2 3
```

- Good R style discourages explicit loops
- In many cases, a loop is the natural way to do it
- Vectorized functions (`apply`, `lapply` and `sapply`) can lead to significant speedups

Loading Data

- Writing vectors and matrices by hand is good for 15 minutes!
- Easier to read data from files
- R supports a variety of text and binary formats
- Certain packages can be loaded to handle special formats
 - ▶ Read from SPSS, Stata, SAS
 - ▶ Read microarray data, GIS data
- Excel files can also be read (easiest on Windows)
- Data can be accessed from SQL databases (Postgres, MySQL, Oracle)
- We'll consider ordinary CSV files

Reading CSV Data

```
read.table(file, header = FALSE, sep = ",", quote = "\"'",  
  dec = ".", row.names, col.names,  
  as.is = !stringsAsFactors,  
  na.strings = "NA", colClasses = NA, nrow = -1,  
  skip = 0, check.names = TRUE, fill = !blank.lines.skip,  
  strip.white = FALSE, blank.lines.skip = TRUE,  
  comment.char = "#", allowEscapes = FALSE, flush = FALSE,  
  stringsAsFactors = default.stringsAsFactors(),  
  encoding = "unknown")  
  
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",  
  fill = TRUE, comment.char="", ...)
```

- `read.csv` is just a specialized version of `read.data`
- In general the arguments you will focus on are `file` and `header`
- The function will return a `data.frame`, with column names (if a header row is available)

Reading CSV Data

- Consider the extract of a CSV file below

```
ID,Class,Desc1,Desc2,Desc3,Desc4,Desc5  
w150,nontoxic,0.37,0.44,0.86,0.44,0.77  
c49,toxic,0.37,0.58,0.05,0.85,0.57  
s97,toxic,0.66,0.63,0.93,0.69,0.08
```

- Key features include
 - ▶ A header line, naming the columns
 - ▶ An ID column and a Class column, both characters
- Reading this data file is as simple as

```
> dat <- read.csv('data1.csv', header=TRUE)
```

- If you print dat it will scroll down your screen

- To get a quick summary of the data.frame use the `str()` function

```
> str(dat)
'data.frame':  100 obs. of  7 variables:
 $ ID    : Factor w/ 100 levels "a115","a180",...: 83 7 66 85 64
 $ Class: Factor w/ 2 levels "nontoxic","toxic": 1 2 2 1 2 1 1
 $ Desc1: num  0.37 0.37 0.66 0.18 0.88 0.37 0.72 0.72 0.2 0.68
 $ Desc2: num  0.44 0.58 0.63 0.09 0.1 0.14 0.03 0.15 0.65 0.66
 $ Desc3: num  0.86 0.05 0.93 0.57 0.85 0.67 0.99 0.68 0.77 0.7
 $ Desc4: num  0.44 0.85 0.69 0.85 0.58 0.17 0.95 0.19 0.69 0.6
 $ Desc5: num  0.77 0.57 0.08 0.17 0.74 0.56 0.22 0.19 0.91 0.1
```

What is a Factor?

- The factor type is used to represent categorical variables
 - ▶ Toxic, non-toxic
 - ▶ Active, inactive
 - ▶ Yes, no, maybe
- They correspond to enum's in C/Java
- A factor variable looks like an ordinary vector of characters
- Internally they are represented by integers
- A factor variable has a property called the levels
 - ▶ Indicates what values are valid for the factor

```
> levels(dat$Class)
[1] "nontoxic" "toxic"
```

What is a Factor?

- If you try to assign an invalid value to a factor you get a warning

```
> dat$class[1] <- 'Yes'
Warning message:
In '[<-.factor'('*tmp*', 1, value = "Yes") :
  invalid factor level, NAs generated

> dat$class[1:10]
 [1] <NA>      toxic      toxic      nontoxic toxic      nontoxic nont
 [9] nontoxic toxic
Levels: nontoxic toxic
```