Whirlwind R notes for STRANGE

Ben Bolker

May 16, 2005

1 why R?

The standard list:

- it's an advanced stats package — comparable to SAS etc., you are very unlikely ever to find a statistical procedure you can't do in R
- complete programming language
- good graphics (although not easy)
- free, both "as in beer" ($) and philosophically (open source)
- cross-platform
- encourages *repeatable* and *automated* data manipulation, analysis, plotting

Disadvantages: speed (vs. C/Java/FORTRAN/MATLAB?); user-friendliness (vs. JMP, Excel, SPSS); handling enormous data sets (vs. SAS; but use database backend); particular problems (vs. MARK, DISTANCE).

2 Getting in and getting out

Lebanese proverb: "when entering, always look for the exit"

Use `q()` to quit (not `q`, which will list the function for you! Say "yes" to saving the workspace if you want to continue on the same problem later.

The Escape (`ESC`) key will stop a running computation or print-out.

3 Interactive calculations

When R is launched it opens the **console** window This has a few basic menus at the top; check them out on your own. The console window is also where you enter commands for R to execute *interactively*, meaning that the command is executed and the result is displayed as soon as you hit the `Enter` key. For example, at the command prompt `>`, type in `2+2` and hit `Enter`; you will see

```
> 2 + 2

[1] 4
```

To do anything complicated, you have to *assign* the results from calculations to a variable, e.g.

```
> a = 2 + 2
```

The variable `a` is automatically created and the result (4) is stored in it, but nothing is printed. This may seem strange, but the default is to *not* to print results so large lists won't fill the screen. To print the value of a variable, just type the variable name by itself

```
> a

[1] 4
```

In this case `a` is a *numeric vector* (of length 1), which acts just like a number: we'll talk about data types more shortly.

You can break lines **anywhere that R can tell you haven't finished your command** and R will give you a "continuation" prompt (+) to let you know that it doesn't thinks you're finished yet: try typing

```
a=3*(4+
5)
```

to see what happens (this often happens e.g. if you forget to close parentheses).

Variable names in R must begin with a letter, followed by alphanumeric characters. Long names can be broken up using a period, as in `very.long.variable.number.3`, but (beware!) you **cannot** use blank spaces in variable names. R is case sensitive: `Abc` and `abc` are **not** the same variable. Make names long enough to remember, short enough to type. Avoid: `c`, `l`, `q`, `t`, `C`, `D`, `F`, `I`, `T`, which are all built-in functions or hard to distinguish.

Calculations are done with variables as if they were numbers. R uses `+, -, *, /, and ^` for addition, subtraction, multiplication, division and exponentiation, respectively. For example:

```
> x = 5
> y = 2
> z1 = x * y
> z2 = x/y
> z3 = x^y
> z2

[1] 2.5
```

```
> z3

[1] 25
```

Even though the values of `x` and `y` were not displayed, R remembers that values have been assigned to them. Type `x` or `y` to display the values.

You can edit commands to correct or modify them. The ↑ key (or `Control-P`) recalls previous commands to the prompt. For example, you can bring back the third-from-last command and edit it to

```
> z3 = 2 * x^y
```

(experiment with the ↓, →, ←, `Home` and `End` keys).

You can combine several operations in one calculation:

```
> A = 3
> C = (A + 2 * sqrt(A))/(A + 5 * sqrt(A))
> C

[1] 0.5543706
```

R also has many built-in mathematical functions: `log()` and `exp()` (and `log10()`), `sin()` and `cos()`, etc..

Exercise: the equation for the standard normal distribution is $\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$. Compute the value for $x = 1$ and $x = 2$.

Logical operators produce `TRUE` or `FALSE` as answers: `==` (double-equals), $>$ and $<$ compare two values, `|` (or), `&` (and), and `!` (not) modify other logical values.

```
> A == 3

[1] TRUE


> A > 2

[1] TRUE


> (A > 2) & (A < 2)

[1] FALSE


> (A > 2) | (A < 4)

[1] TRUE
```

```
> !(A > 2)

[1] FALSE
```

Exercise: convince yourself that `!(a & b)` is equivalent to `!a | !b`, no matter what the values of `a` and `b` are.

4 Data types and structures

4.1 vectors

Lists of values, all the same type or *class* (numeric, logical, character). As mentioned above, a number is just a vector of length 1. Some functions create vectors: you can also create/assign vectors by, e.g.

```
> x = c(1, 2, 7, 8, 9)
```

Typing `1,2,7,8,9` without `c()` gives an error. Vector elements can be named:

```
> x = c(first = 1, second = 1.2, third = 1.5)
```

Refer to elements in vectors s

- by *position*: `x[1]` or (multiple elements) `x[c(1,2)]`
- by *name*: `x["first"]`
- by *exclusion*: `x[-1]` drops the first element
- with *logical vectors*: `x[c(TRUE,TRUE,FALSE)]` gives the first two elements; more usefully, `x[x>1.1]` first computes a logical vector `[TRUE, FALSE, FALSE]` and then uses it to select the first element only.

Most of R's functions are *vectorized*: give them vectors and they automatically do the right thing. Functions of two vectors (`c(1,3,5,7) + c(2,4)`) will automatically *replicate* the shorter vector until it is as long as the longer one, giving a warning message the longer is not an even multiple of the shorter.

```
> x = c(1, 3, 5, 7)
> 2 * x

[1]  2  6 10 14


> x + c(2, 5)

[1]  3  8  7 12


> x + c(1, 2, 4)
```

```
[1] 2 5 9 8


Warning message:
longer object length
        is not a multiple of shorter object length in: x + c(1, 2, 4)
```

Other functions in R are inherently vector functions: `mean()`, `var()`, `sum()` ...

4.2 Matrices

Matrices are tables of data, all of the same type (numeric, character, logical).

Retrieve elements from matrices, selecting rows, columns, or both, by placing commas appropriately. Rows come first (before the comma) and columns second. `p[1,]` (row 1), `p[,2:5]` (columns 2 through 5), `p[3,4]`.

Create a matrix with `matrix()`: by default R orders matrices *column-first*:

```
> matrix(c(1, 3, 4, 5), nrow = 2)

     [,1] [,2]
[1,]    1    4
[2,]    3    5


> matrix(c(1, 3, 4, 5), nrow = 2, byrow = TRUE)

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

Matrices act like vectors when appropriate:

```
> log(matrix(c(1, 3, 4, 5), nrow = 2))

         [,1]     [,2]
[1,] 0.000000 1.386294
[2,] 1.098612 1.609438
```

4.3 Lists

Lists are collections of *anything*: vectors and matrices of different sizes and types, results of statistical analyses, etc.. Use `$` to pull out the elements of a list by name, and `[[]]` to pull out elements by number.

```
> x = list(x = c(1, 2, 3), y = c("a", "b"))
> x[[2]]
```

```
[1] "a" "b"
```

```
> x$y
[1] "a" "b"
```

4.4 Data frames

Data frames are a confusing but useful cross between lists and matrices: lists of equal-length columns, which can be different types, and which are treated like either matrices or lists depending on the context. You can access their elements either like a list (`x$x, x[[1]]`) or like a matrix (`x[2,4]`).

5 `rep()` and `seq()`

These two functions are very useful for generating vectors: `seq()` (and its abbreviation, :) generates different kinds of regular sequences, while `rep()` replicates existing vectors.

```
> rep(1:4, 3)
 [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

```
> rep(1:4, each = 3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

```
> rep(1:4, c(1, 2, 3, 4))
 [1] 1 2 2 3 3 3 4 4 4 4
```

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

```
> seq(1, 10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
> seq(1, 10, by = 2)
[1] 1 3 5 7 9
```

```
> seq(0, 2, length = 7)
[1] 0.0000000 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000
```

6 Getting help

- Typing `?` followed by the name of a function (e.g. `?mean`) will pop up a window with information on the function (only useful if you already know its name (!) and it is part of a package that has been loaded
- `help(package=packagename)` lists and gives brief descriptions of all of the functions in a package
- `help.search("word")` looks through all documentation, including installed but unloaded packages, whose name or short description includes `word`. *Does not do full-text search*, nor have R functions been extensively cross-indexed, so you have to be close or get lucky.
- `help.start()` (with parentheses) pops up a browser window: go to Packages (probably `base`, `stats`, or `graphics` to find info on functions
- `RSiteSearch("word or phrase")` (new!) goes to the R web site and does a query
- `example("function")` runs the examples given in the help page for the function

Ask an instructor!

7 Getting stuff in and out of R

7.1 Data

Basic input functions are `read.table()` and `read.csv()` — see other notes.

Save objects in R format with `save("x","y","z",file="savefile.RData")`. Write data out to a file with `write.data("mydata",file="mydata.txt")`.

7.2 Code

Your own code, written in a text file: `source("myfuns.R")`. Pre-existing code that has been compiled in a `package` and installed on your system (including functions found by `help.start()`): `library(packagename)`. Installing new packages on your system: `install.package(packagename)`.

For smaller pieces of code, you can simply cut and paste from Notepad or Wordpad, Tinn-R, or another *text editor* such as emacs. **Do not use Microsoft Word to edit R code, it will screw things up.**

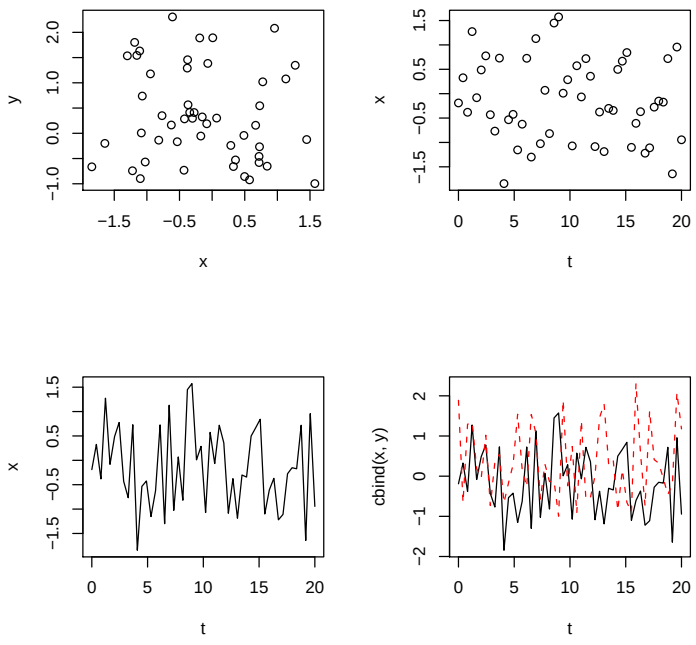
Save code by cutting and pasting (also possibly `save()` or `dump()`).

8 Plotting

```
> t = seq(0, 20, length = 50)
> x = rnorm(50)
> y = rnorm(50)
```

I will explain `par(mfrow=c(2,2))` in a bit: it makes a 2×2 array of plots on the page. plot array:

```
> par(mfrow = c(2, 2))
> plot(x, y)
> plot(t, x)
> plot(t, x, type = "l")
> matplot(t, cbind(x, y), type = "l")
```



`plot()` is a generic function that *may* do the right thing if you just give it a data object, for example a *factor* (a categorical variable):

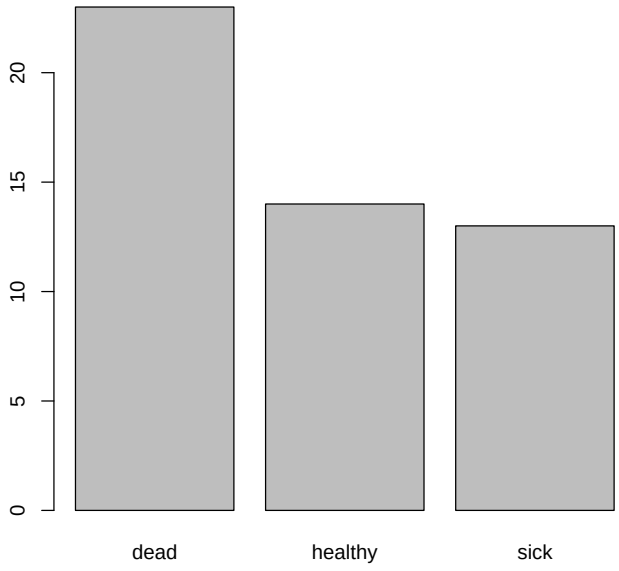
Sample at random (with replacement)

```
> f <- factor(sample(c("healthy", "sick", "dead"), size = 50, replace = TRUE))
> table(f)

f
   dead healthy    sick
```

```
     23      14      13
```

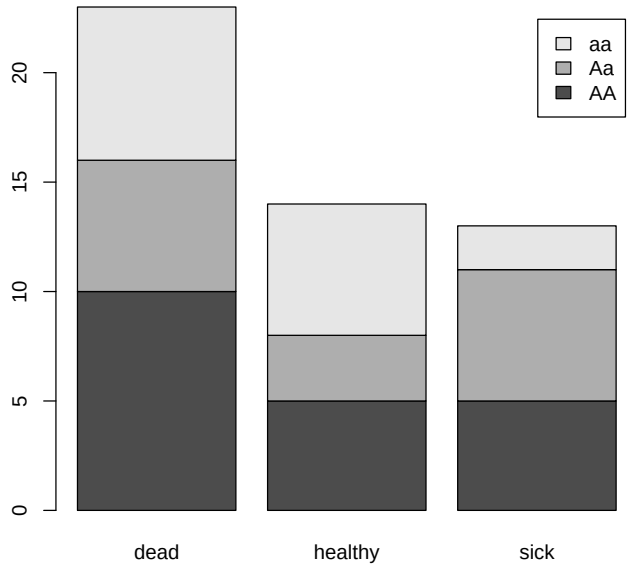
```
> plot(f)
```



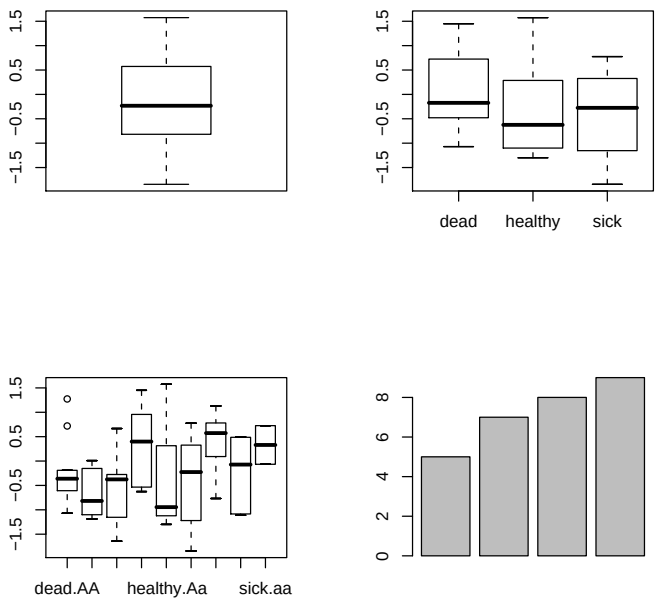

```
> g <- factor(sample(c("AA", "Aa", "aa"), size = 50, replace = TRUE))
> table(f, g)
```

```
         g
f         AA Aa aa
  dead    10  6  7
  healthy  5  3  6
  sick     5  6  2
```

```
> plot(f, g)
```




```
> par(mfrow = c(2, 2))
> boxplot(x)
> boxplot(x ~ f)
> boxplot(x ~ f + g)
> barplot(c(5, 7, 8, 9))
```



There are **many many** options for changing axes, labels, adding legends, colors, line types, line widths. The help page for the `par()` command (`?par`) is voluminous, but you can search within it for the stuff you want to find. In general, `par("name")` queries the current setting of graphics parameter `name`, while `par(name=value)` sets the value of the parameter (you can set several at once: `par(name1=value1,name2=value2)`). `par()` shows *all* the parameters at once.

9 Flow control

- `if (condition) { } else { }` (braces around chunks of code)
- `elseif(x,y,z)` returns `y` if condition `x` is true, otherwise (else) returns `z`: e.g. `ifelse(x<0,0,x)` (this particular example is equivalent to `pmax(0,x)` where `pmax()` is "parallel (vectorized) maximum")
- `for (i in x) { }`: works through the elements of `x`, setting `i` to each one in turn. Most common is `for (i in 1:n)`. If working through a non-integer vector (e.g. of parameters; `beta=seq(1,10,by=0.1)`, still probably want to do `for i in 1:length(x)` so that you can use `i` as an index for saving the results ...
- `while () {}`

10 Functions

Many **many** built-in functions. *Arguments* are sometimes obvious, e.g. `sin(x)`. More complex functions can have many different arguments (e.g. `matrix`); these can be specified by name or by position:

```
matrix(data=1:12,nrow=2,ncol=3,byrow=TRUE)
```

is the same as

```
matrix(1:12,2,3,TRUE)
```

but the first is obviously easier to understand. Most complicated functions also have *defaults* for most of their arguments, so you can specify just the ones you want to be different from the defaults. Named arguments can be in any order, so `matrix(1:12,ncol=3,byrow=TRUE)` is also equivalent to the variants above.

You can (and should) easily define your own functions: a very simple example is

```
square <- function(x) {
  x^2
}
```

The last statement in the function is its *return value*; you can also use the `return()` function to specify what the sfunction returns. It's a common mistake to forget this:

```
myfun <- function(x) {
  if (x<0) {
     y=x^2
  } else {
    y=x^3
  }
}
```

returns *nothing*: it should be

```
myfun <- function(x) {
  if (x<0) {
     y=x^2
  } else {
    y=x^3
  }
  y
}
```

An equivalent solution:

```
myfun <- function(x) {
  if (x<0) {
     return(x^2)
  } else {
    return(x^3)
  }
}
```

Scope: inside functions, variables are *local*, insulated from the "calling environment". You can re-use variables inside a function that you use outside a function, without changing the value once the function is done. This means *you can't use a function to change the value of a variable.*

Calling the following function

```
squarex <- function(x) {
   x=x^2
   return(x)
}
```

will not change the value of `x`. You need to say `x = squarex(x)`. If you *absolutely must* you can use the global assignment operator `<<-`, but you probably shouldn't.

11 Maximization and fitting

Give `optim()` a function (that takes a vector of parameters and returns a single number, e.g. a goodness-of-fit value), and a vector of parameter starting values, and it will try to find the parameter values that minimize the function.

12 Probability distributions

R knows about most distributions you would ever care about. For each, there are four associated functions that give the probability density function, cumulative distribution function, quantile (inverse CDF) function, and a random-deviate generator. Each has standard parameters for its type and the parameters of the distribution.

13 Standard statistics

- Linear regression, ANOVA, ANCOVA: `lm()`
- Generalized linear models (logistic regression, Poisson regression, etc.): `glm()`
- Survival analysis: package `surv`

- Others: `t.test()`, `chisq.test()`, `pairwise.t.test()`, `binom.test()` (difference in proportions, binomial samples), `cor.test()` (correlations), `wilcox.test()` (Wilcoxon/Mann-Whitney nonparametric), `kruskal.test()` (Kruskal-Wallis), `ks.test()` (Kolmogorov-Smirnov), ...