What is `R` ?

- a statistical *language* (in the same broad sense of `c++` or `python`)
- a statistical *environment* (in the same broad sense of `STATA` and `SPSS`)
- great
- free
- used pretty much exclusively in the department for methods work
- regularly updated: what version are we using? (try typing `version` in a minute)

Get it here: `http://cran.r-project.org/`. It will run on almost anything.

Read more about it here: `http://www.r-project.org/`.

Note: free Manuals and Search facilities. Try an `R` Site Search for 'covariance matrix.' You should see mailing list questions others have asked—a fantastic source of info.

It is maintained by volunteers, and volunteers write packages. There are a lot of these.

`R` does most things very well. It is preferred to almost any other (statistical) environment especially for Bayesian and non-parametric stuff. Graphics are unbeatable (especially at this price).

Open Up Basics

When you open `R` (which you will find via the Start menu), you should see RWinEdt open too. People sometimes use `R` directly, or they write 'programs' in RWinEdt and then paste or source them. If the program contains more than one or two operations, use RWinEdt.

You should also see a list like this:

```
Loading required package: RWinEdt
Loading required package: utils
Loading required package: MASS
Loading required package: grDevices
....
```

`R` is telling you it is automatically loading a set of packages that might be helpful. This is specific to `the star lab`, and is not how factory `R` is distributed.

Note: stuff that `R` has already done is in blue; when it wants something from you, you should see a red command line prompt.

What can `R` do? Try:

```
demo(graphics)
```

Nice enough. Then try :

```
library(rgl)
demo(rgl)
```

Use the mouse to mess around with the plots. Notice that the `R` console is different to the `R` Graphics window: be careful with maximizing etc.

`R` as a calculator

`R` can be used as a calculator. Try `2+3 <Enter>`, `2*3<Enter>` and `2^3<Enter>`—you get the general idea.

Be careful with brackets and braces: `(2+3)/(6-2)`≠ `(2+3)/6-2`.

Guess what `log`, `exp`, `det` do. . .

Getting Help

If you know the name of the function on which you need help (more on this below), use a question mark. Try:

```
?mean
```

and notice the window gives you (very helpful) examples that you can cut and paste.

Suppose you weren't sure of what you wanted, but knew it had something to do with `cov` (say, covariance). Try:

```
apropos("cov")
```

which suggests a bunch of topics connected to that key-word. Use this list to get, say,

```
help("discoveries")
```

and maybe do one of the examples (cut and paste).

Assignments & Vectors

The basic `R` operation is the assignment, and its operator is `<-` which is formed by `shift+,` and `shift+_` (next to the zero key). We can pretty much assign anything to anything. Try this:

```
a<-34
```

Now `a` has all the 'properties' of 34. For example, try `a+5`. Say I want `b` to have the same properties...

```
b<-a
```

will take care of that. Now, try `b+a`: exactly as expected! Everything we've said here applies to characters (rather than numbers) too:

```
d<-"monkey"
```

but notice that we have to use quote marks. You can't assign a value to an object that begins with a number, so

```
5f<-59
```

won't work, but `f5` will work fine.

Vectors are keys building blocks (of course, the assignments above are just vectors of length 1). To make a vector, `g` we use the concatenate command, written as `c`. So, try

```
g<-c(3,4,7,8)
```

Again, try adding 5 to `g`, and you'll see it goes element by element. Character vectors are also possible:

```
animals<-c("elephant", "giraffe","zebra", "goat")
```

But obviously `R` won't appreciate you wanting to add 5. We can mix it up:

```
some.stuff<-c("elephant", "giraffe","zebra", "goat",78)
```

Notice that the individual members of the vectors can be pulled out with straightforward use of square braces. If you want the first element of `g`, try `g[1]`. This indexing differs from some other languages. If you want the second and third elements of `animals`, try `animals[c(2,3)]` (note the use of the concatenate!). A couple of extra helpful things here:

1. we can get the last element of the vector by using the fact that it occurs in the same position as the length of the vector itself. So `g[length(g)]` is really `g[4]`... which is 8.
2. we can use `from:to` syntax to make life easy: say you want the first through third elements of `animals`. One way is `animals[c(1,2,3)]`, but a better way is `animals[1:3]` (no need for concatenate now). If animals was, say, 10000 elements long, you would appreciate the time saved.

A final thing: notice that we can assign vectors 'to themselves.' Consider:

```
j<-c(1,2,3)
j<-c(j,4)
j<-j+1
```

This can be very helpful sometimes (but very confusing at other times!).

Functions

There are literally thousands of functions in `R`, and people are always writing more (including you in 404/405/505/506). Let's say you wanted to sort the vector `g` above:

```
sort(g)
```

will do it. Several things to notice here:

1. the function `sort` works by placing round braces around the thing you want it to operate on (no weird spaces, unlike `STATA`).
2. you can get help by typing `?sort`
3. notice from the help that `sort` has a whole series of *arguments*. The function has *defaults* which are given in the `Usage` box in the help. We can specify that `R` sort in a different way by altering these arguments: try `sort(g, decreasing=T)` where `T` stands for true.
4. we can either name the argument, or we can match its position in the call. More on this later...

Try and sort the `animals` vector too: it will put everything in alphabetical order. In your own time, take a look at `?order` which can be very helpful sometimes.

Matrices

We can write matrices from scratch. Try

```
mat<-matrix(c(1,3,5,2,7,3,2,4,4),nrow=3,ncol=3)
```

We tell `R` we want the matrix to consist of the numbers in the vector, and that it should be 3 rows by 3 columns. Notice that `R` will fill the matrix *by column*. Often we want it filled *by row*. Let's try again:

```
mat<-matrix(c(1,3,5,2,7,3,2,4,4),nrow=3,ncol=3,byrow=T)
```

where the `T` tells `R` to behave slightly differently. For more information, take a look at `?matrix`. The transpose of a matrix is obtained by `t()`, and the inverse is `solve()`. We can get the eigenvalues/vectors using `eigen()`.

Unsurprisingly, we can bind vectors together to make matrices. Define a couple more vectors (in addition to `g` above):

```
h<-c(4,5,3,2); i<-c(9,0,4,1)
```

Notice the use of `;` which simply tells `R` I want another command on the same line (you could have used a return instead). We can bind our vectors by row (`rbind`) or column `cbind`. Take a look at

```
cbind(g,h,i)
```

and

```
rbind(g,h,i)
```

Of course, we would generally assign the `cbind()` or `rbind()` of these vectors to some new object, e.g.:

```
mat.2<-cbind(g,h,i)
```

Much as with vectors, we can pull out cells from our matrices. Now the syntax is `mat.2[row, column]`. So

```
mat.2[3,c(1,3)]
```

will return row 3, columns 1 and 3 of `mat.2`. Notice that we can use the `-c()` command to drop rows and columns. So:

```
mat.2[-1,]
```

will drop the first row of `mat.2`

Random Numbers

Suppose we want 100 random numbers drawn from a normal distribution, with mean 2, standard deviation 5. In general, we can use `r`*distribution name*. For the normal, we can use `rnorm`, for the beta, we can use `rbeta`, for the poisson it's `rpois` and so on. The syntax you need can be found via the help functions (e.g. `?rnorm`), but in our case:

```
norm.nums<-rnorm(100, mean=2, sd=5)
```

will do the trick. We can get 102 beta random numbers with $\alpha = 3$, $\beta = 6.2$ with

```
beta.nums<-rbeta(100, shape1=3, shape2=6.2)
```

Notice that `rbeta` doesn't use intuitive naming for the arguments, and make sure to read the help files to check how the parameters of the distribution are defined.

An alternative way to generate random numbers is take a series of quantiles between zero and one, and then push it through a density transformation. This is a bit inefficient—in that it is implicitly discrete, and assumes you 'know' something about the support of your distribution—but works nonetheless. So, try

```
ruler<-seq(-5,11,length=1000)
norm.nums.2<-dnorm(ruler,mean=2,sd=5)
```

Notice that `norm.nums.2` is not a set of random normal values in the sense that `norm.nums` is. Rather it is a density for every point in `ruler`.

Plotting, lines and Histograms

`R` is renowned for its fantastic graphics.

The key command is `plot()` which takes a possibly very large number of arguments, but for our purposes:

```
plot(x, y, main=" ", xlab=" ", ylab=" ")
```

For right now, create `L<-rnorm(100)` and `P<-rnorm(100)`. Now try

1. `plot(L,P)` which treats the object `L` as the thing on the x-axis and `P` as the thing on the y-axis (notice also that we are using 'positional' matching in the `plot` call.
2. it doesn't have a title right now, so we need to redo our call to `plot()`. But notice: every time we call `plot()`, we redraw the graphic! So:

   ```
   plot(L,P, main="My plot", ylab="The y axis", xlab="The x axis")
   ```

   I switched the position of the `ylab` and `xlab` call, but that makes no odds because the matching is being done by name (not position) here. Make sure to use quote marks for title names!

   Sometimes we want a line plot, not a point plot. Consider:

   ```
   plot(1:length(P),P, main="My plot", ylab="The y axis", xlab="The x axis")
   ```

   but then try:

   ```
   plot(1:length(P),P, main="My plot", ylab="The y axis", xlab="The x axis",type="l")
   ```

   and guess what `type="l"` does (hint: `"l"` is short for `"line"`).

We sometimes want to more lines or points on a plot after we have drawn it. This will generally be accomplished with `lines(x=, y=,lty=)` or `points(x=, y=)`. Try

```
lines(x=1:length(P), y=P-1,lty=3)
```

which will draw a line of type 3 (`lty=3`, which is a broken line) linking the x-axis as is to $P - 1$ on the y-axis. Notice that the original `plot` call set the size and nature of the axes (so some of the new lines appear off the original plot).

A very useful function is `plot(density())` which allows us to plot a (kernel smoothed) density. Try:

```
plot(density(P))
```

Histograms take little extra imagination:

```
hist(P)
```

Sometimes you want to have a histogram and an imposed smoothed density, in which case you need to tell `hist()` not to give back frequencies:

```
hist(P, freq=F)
lines(sort(P), dnorm(sort(P), mean=mean(P),sd=sd(P))
```

Notice that we have to sort `P` for the overlay to make sure the line runs left to right.