R-WinEdt

Open `R` and RWinEdt should follow: we'll need that today.

Cleaning the memory

At any one time, `R` is storing objects in its memory. The fact that everything is an object in `R` is generally a good thing, but you need to be careful. Try

```
ls()
```

to get a list of everything that's there now. You may be surprised to see lots of stuff; in fact, it has been produced by those packages loading. Get rid of it before we begin:

```
rm(list=ls())
```

As good practice, put this command at the top of *every program you write*.

While we're at it, notice that `R` is working in a particular directory right now. To get the working directory, use

```
getwd()
```

This probably doesn't matter much today, but it can be important when you want to work from a specific place on your `z:` drive. To set the working directory, use `setwd()`.

Commenting and Other Conveniences

We can comment code using `#`:

```
#this is a comment
f<-c(4,5,2)
#this is a comment also
```

will create the object `f`... and nothing else. Use `#` wisely and liberally in your own programs.

If not already obvious, use the uparrow to get the previous command back.

Use a semi-colon between commands to get several on one line:

```
a<-2; d<-4; j<-100
```

Reading Data

We often want to read in data, but it's format varies. The most common situations are reading in `.txt` files, `.csv` files and `.dta` (i.e. `STATA`) files.

To see how we would use the commands, go to `http://mail.rochester.edu/ spln/R/data/` and download the data to your `z:` drive.

We will read the data into `R` (using slightly different calls each time) and then look at it.

Text files/Tables/ASCII

Start with

```
football<-read.table("z:/path/path/football.txt", header=T)
```

where, obviously, `path` refers to wherever you put it. Notice:

- `read.table()` is generally what we use for `.txt` files
- get help using `?read.table`
- the pathway is not case sensitive
- use forward slashes in the pathway
- `header=T` tells `R` the first line is a 'header' with column names—try reading the data in again *without* this argument
- the new object is called `football`: we could have called it anything we liked

An important point about `.txt` data is that we need to have the same number of entries in every row. This can be a problem when, say, names of things have spaces. Say you had a file with a set up like this:

```
City        Population
Houston    2144491
Dallas      1232940
San Antonio     1296682
Austin      709893
```

Although you don't need to have all the entries *aligned* with the columns, you do need to make sure you have one *and only one* entry per column. Here, `R` would complain about the third line: it has three entries, but only two columns (or `R` might complain that the other columns don't have enough entries).

We can type `football` to see it, but it is generally better to use

```
edit(football)
```

which gives an 'active' window. Even better, try

```
library(relimp)
showData(football)
```

CSV files/comma separated/excel

These files are comma separated, and you should be able to open them directly in excel (try it).

Bring the `.csv` file into `R` using something like

```
UK.data<-read.csv("z:/path/path/UKtime.csv")
```

Again, try looking at is via `edit()`.

STATA file/`.dta`

STATA files are generally in a `.dta` format. `R` needs to load a package before we read these in though:

```
library(foreign)
```

Actually, this package will allow us to `read.spss()` as well. We can read in the `.dta` file using

```
coal.data<-read.dta("z:/path/path/coal.dta")
```

Data Properties

For various reasons, you will often want to know your data's dimensions. Try:

```
dim(coal.data)
nrow(coal.data)
ncol(coal.data)
```

You might also want the variable/column names:

```
colnames(coal.data)
```

should take care of that. Though we typically conceive of this data as a matrix, in fact it is actually being stored here as a *data frame*. In practice, this means that `R` is doing various additional operations like converting various columns to factors. We'll come back to this.

Notice that, if we have the column names, we can pull any vector of data we are interested in using the `$` sign:

```
coal.data$fract
```

Literally, we are asking for the part of `coal.data` with the name `fract`. We could also, of course, use the numerical referencing system we learnt before. So, try:

```
coal.data[,27]
```

which will pull the 27th column.

Summaries are easy to come by, and often useful:

```
summary(coal.data$fract)
hist(coal.data$fract)
plot(density(coal.data$fract))
```

Attaching and the Search Path

It's annoying to keep using `$` every time we want a variable.

An alternative is to use `attach()` which sticks the data frame in `R` 's search path. To see `R` 's current search path use

```
search()
```

This can be helpful when you have variables in different data sets with the same names. So

```
attach(coal.data)
```

will attach the coalition data. Now

```
polar
```

will spit back the `polar` variable. Use `detach(coal.data)` to get rid of the data from the search path. If you just use `detach()`, without an argument, the object in the second position of the search path (which, here, will be your data) will be dropped.

Editing/Writing data out

`R` was not designed for large scale data management and editing. Facilities exist to do it, of course, but they can be a little clunky. Nonetheless, try:

```
football2<-edit(football)
```

and change something in the data. Now,

```
edit(football2)
```

should be an edited version of that data set. What many people do instead of this is write the data out to another program (esp excel) and edit it there. So, try

```
write.csv(football, file="z:/pathway/football2.csv")
```

where `pathway` is where you are keeping your data. Now open excel and find `football2.csv`. You should be able to edit it freely.

Properties of Things

Above, we learnt a little about `R` thinks about data: its dimensions, number of rows and so on. But we can be broader about this idea: we might want to ask `R` about any object.

One way to do this is to ask what the `mode` of an object is. First, create some objects:

```
a<-c(8,7,34, pi)
b<- c("gold", "sword", "juno")
```

Then try:

```
mode(a)
mode(b)
```

A better—more descriptive—way is to use `str()`. Try:

```
str(a)
str(b)
str(football)
```

We can also ask `R` directly using a ‘logical’: try

```
is.matrix(football)
is.vector(football)
is.data.frame(football)
```

For data work, it’s important to bear in mind the difference between variables that are numeric, those that are characters, and those that are factors. To see the difference, load up some of the built-in data, and then take a look at one of the variables

```
data(iris)
iris$Species
str(iris$Species)
```

This variable is a ‘factor.’ This is different to a numeric or a character vector. Factors specify a *discrete grouping* of their components. Here, it is the Species of iris flowers: these are not numeric, but they are data and can be used in a regression or other procedure (i.e. they are not simply characters). The simplest way factor is a dummy: say sex which has ‘male’ or ‘female’ as its *levels*.

To make a factor, use `factor()`. Try

```
str(coal.data$prox)
var<-factor(coal.data$prox)
str(var)
```

If you want to convert a factor to a numeric vector (which comes in handy sometimes) you need an extra step—use `as.numeric(as.character( ))`.

```
var2<-as.numeric(as.character(var))
str(var2)
```

and we're back to where we started…

Tables

Tables are pretty straightforward. Try

```
data(quakes)
qtable<-table(quakes$stations)
qtable
```

for fun (note the assignment: not strictly necessary, but handy in a moment). In fact, this is a table of the number of stations reporting 1000 seismic events near Fiji since 1964. So, there were 20 instances of one station reporting, 28 instances of 11 stations reporting, 1 instance of 132 stations reporting etc. What is often helpful in these situations is

```
plot(table(quakes$stations))
```

To get a proportions table, there is a little work-around:

```
table(quakes$stations)/sum(table(quakes$stations))
```

A couple of last things:

- `names(qtable)` will give you the 'names' (in this case categories) that the counts are binned into
- `as.numeric(qtable)` will give you the actual values in the table

Saving

The core functions you'll need are

- `save(x, file="z:/path/xyz.rdata")` which saves the object `x` to the file `xyz.rdata`. You can get it back (another time) using `load("z:/path/xyz.rdata")`. When you ask for `ls()` it will tell you that `x` (whatever that is) is available. Try it with the coalition data.
- `save.image()`. When you quit `R` , you are asked if you want to save the workspace image. This means saving *all* the objects that are currently sitting in `R` 's memory. This time, use `.Rdata` as the extension. So, `save.image(file="z:/pathway/RstuffSession1.Rdata")` will put everything into a file called `RstuffSession1.Rdata` which you can later load…and get all your objects from.

I personally think it is better, when possible, to write (commented) *code* that will create the objects from scratch. . . and then save that. This way, you won't have to try and remember where and/or what everything is: you can just make it again.