

BirminghamMDIBL

Andrew P. Beckerman, University of Sheffield

28 February 2016

Part I starts here....

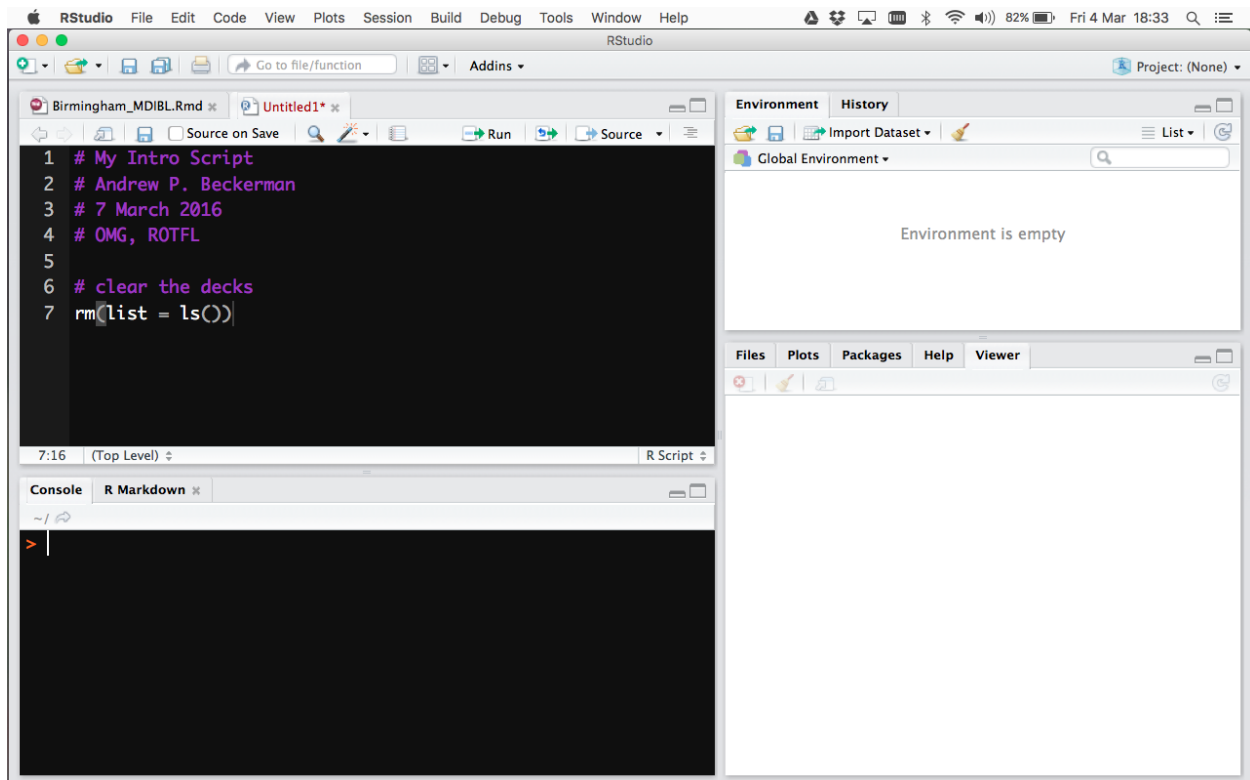
Using R with Rstudio is AWESOME.

Note that they are different. R is the engine. The programming language. The be-all and end-all. Rstudio is a wrapper around R. The maker it a bit betterer.

R and RStudio allow building an annotated, archived, repeatable, share-able cross platform record of what you do.

4 panes:

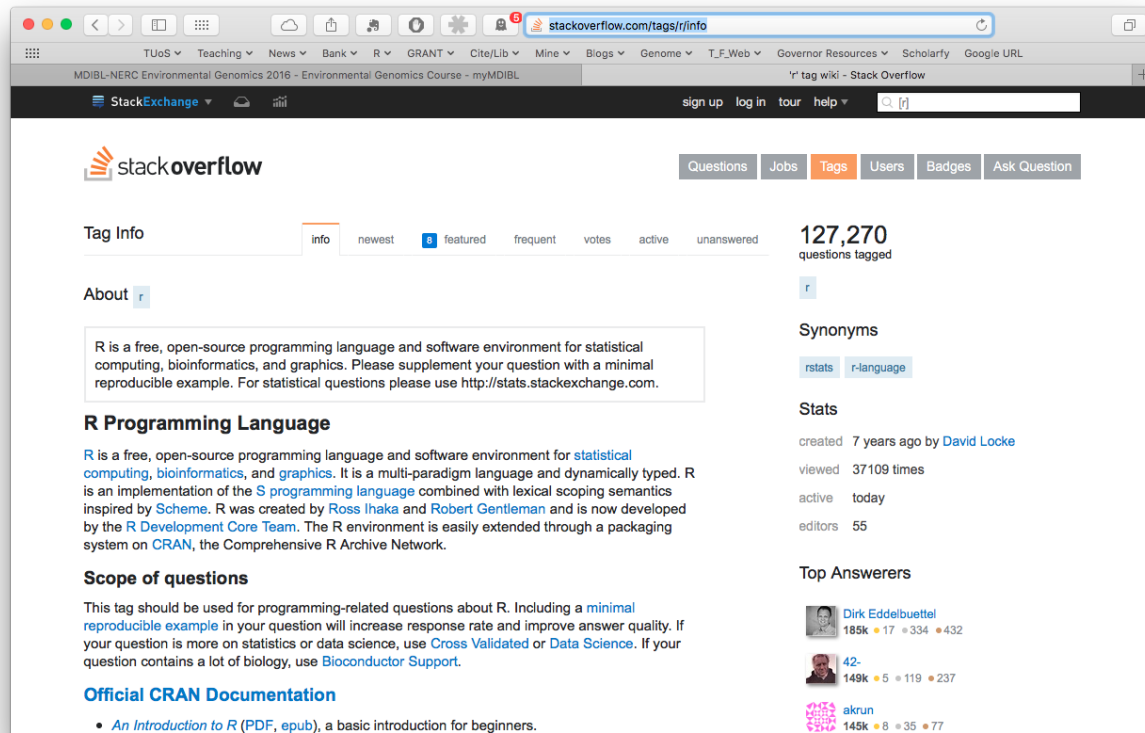
- the console (engine)
- the script (your code)
- Plots/Packages
- Data Stuff



Getting Help

There are many ways. Inside R, we use `?` and `??` or `help.start()`, which spawns a web browser. Outside R, we use... google! But there is something MUCH better. Stackoverflow. This has an R channel, and here you will find VERY constructive answers to questions, including code and graphs. It has rapidly become the go-to location for help. Lots of R developers are active there.

[LINK to StackR](#)



Using R as a calculator

R is a big giant calculator. Lets practice in the CONSOLE. Type each of these, and press ENTER. As you do these, WATCH what RStudio does with `()`'s

Practical

```
# -----  
# BASIC PRACTICAL  
# -----  
# Maths and Functions  
1+1  
2*7/8-9  
log(exp(1))  
log10(1000)  
log(1000)
```

```
sin(2*pi)
2^10
sqrt(81)
```

What you should know from this.

- R does maths correctly
- the default logarithm is the natural log, ln, not base 10
- R has built in function and pre-defined objects

Try the help - use `?log` to see the help file (lower right) for log...

```
# -----
# Try the ?
# -----
?log
```

Moving to the script.

Lets start using the script now. Add some basic annotation at the top, a place holder for libraries (see below) and a good practice activity - clearing R's brain. Note that the `#` symbol is special. It designates the verbiage after it as *comment* or *annotation*. R will NOT try and interpret it. It is INFORMATION FOR YOU.

```
# NAME
# DATE
# Intro to R script

# libraries I need to use

# clear R's brain
rm(list = ls())
```

Now, lets have some more fun... building sequences of numbers to see how R works. Add each of these lines of code to your script. Once you've added them, try this. Simply place the flashing cursor on the line with `1:10`. Now, press **CTRL - Enter**. This should send the code from the script to the console. Don't forget to save your script too!!!

```
# Sequences, Vectorisation
1:10
seq(from = 1, to = 10, by = 1) # note the three arguments, from, to and by
seq(from = 1, to = 10, length = 12) # note the three arguments, from, to and length
```

What you should know from this.

- R can make sequences of integers easily, via :
- `seq()` is short for *sequence*. R is like that.
- `seq()` is a function that takes at least three arguments.

- from, to, by is a sequence in “steps” specified by *by*
- from, to, length is a sequence of a fixed number of numbers, specified by *length*.
- Or, it is a vector with *length* numbers between *from* and *to*
- CTRL - Enter sends info from the script to the console.
- # is for comments and annotation.

Assignment of numbers, vectors and data to *objects*

You can make objects in R. We call this ASSIGNMENT, and we typically use <- (not =, though it is possible). Add these to your script and Run each one.

```
# assignments
x <- 1:10

# look at it (e.g. press CTRL -ENTER)
x

# use it to make another variable
y <- x^2

# use both operations in additional operations.
# Look at the output carefully!
x+y
x*y
```

What you should know from this.

- You can create objects to use.
- The objects can be vectors, matrices, data frames etc.
- R can perform operations with objects.
- operations are by default *element - by - element*
- there is facility for matrix multiplication and linear algebra

Making some graphics. . . . BASE graphics

Here we introduce the basics of making a plot. There are several ways to make pictures. Some pretty good stuff is built into R. The most *popular* now is ggplot2, which is an add-on. But there is also the lattice library. More on packages and libraries below.

To understand how to make a plot, we need to make some data. Then, we roll these data in to a data frame, which is like a spreadsheet.

```
# Your first DataFrame
x <- 1:10
y <- x^2

# Make a data.frame from this
myDat <- data.frame(Beer = x, Goggles = y)
myDat
```

```
##      Beer Goggles
```

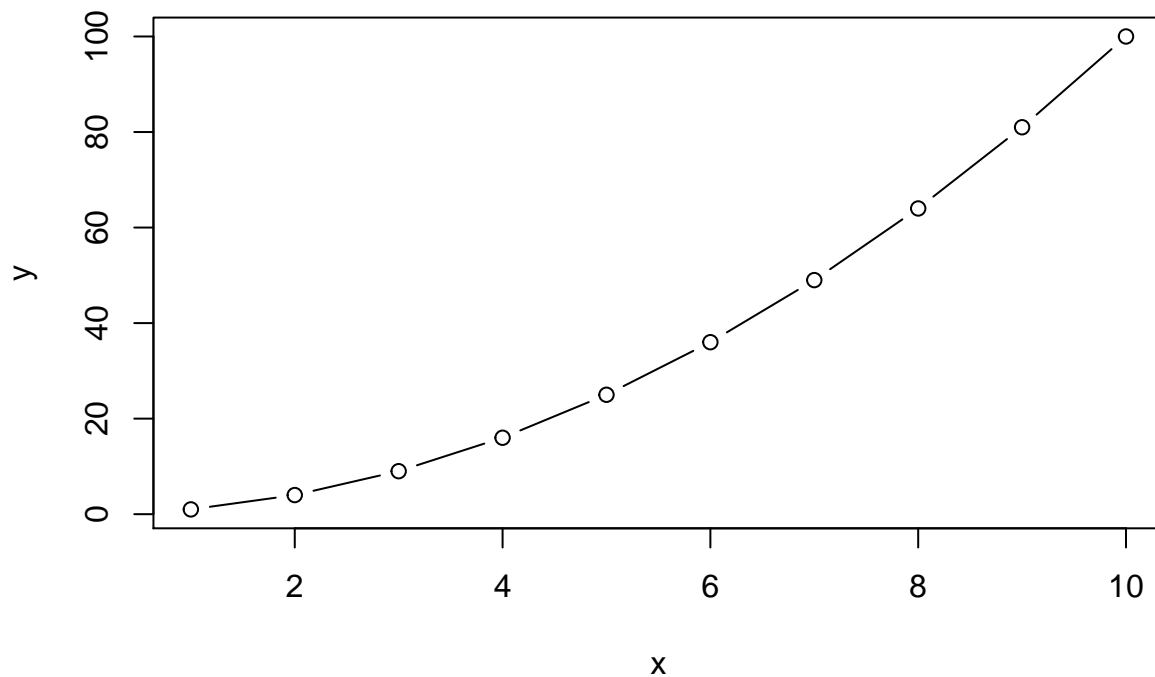
```
## 1      1      1
## 2      2      4
## 3      3      9
## 4      4     16
## 5      5     25
## 6      6     36
## 7      7     49
## 8      8     64
## 9      9     81
## 10     10    100
```

What you should know from this.

- data.frame makes a spreadsheet like object with columns
- we name the data frame using <-
- we name the columns inside the data.frame using =

Right...lets make the picture! We use the base function `plot()` and a *formula* that specifies the y and the x axes of the plot.

```
# Your first plot
# FORMULA INTERFACE
plot(y ~ x, data = myDat, type = "b")
```



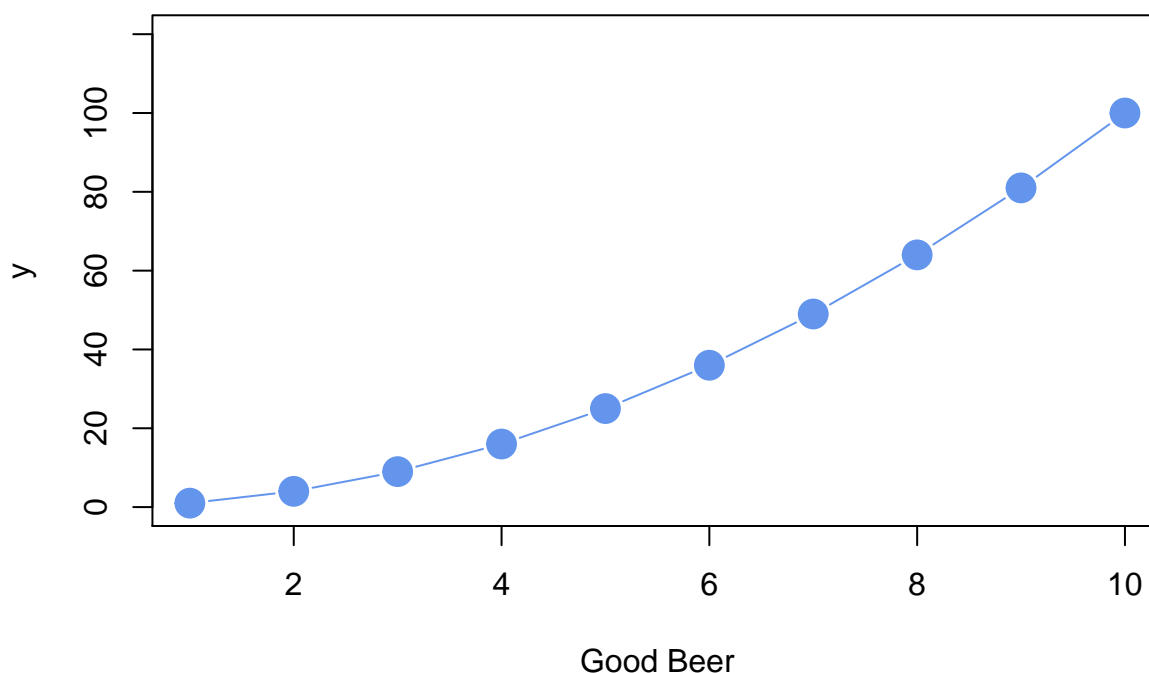
What you should know from this.

- plot is nice
- when the data frame has column names that are nice, they look nice on the graph
- the `type` argument *b* is for both. *p* is for points only. *l* is for lines only.

Semi-ADVANCED Plots

Plots can be customised. ... here we alter the limits of the y-axis (`ylim`), the label on the x-axis (`xlab`), the size points (character expansion: `cex`), and make the points filled (point character; `pch`) with cornflowerblue-ness (`col`) and note the `c()` is a collector...`ylim = c(0,120)` specifies that the lower and upper limit of the y-axis is 0 and 120.

```
# FORMULA INTERFACE
plot(y ~ x, data = myDat, type = "b",
     ylim = c(0,120),
     xlab = "Good Beer",
     pch = 19, cex = 2, col = "cornflowerblue")
```



Back to some Mechanics... Managing Packages

In your 'omics life, you will need to access packages from TWO locations. CRAN, which is the *Comprehensive R Archive Network*, and Bioconductor.

CRAN has ~7500 packages available and they are curated at the CRAN Task Views website:

Packages: [LINK](http://star-www.st-andrews.ac.uk/cran/web/packages/) `http://star-www.st-andrews.ac.uk/cran/web/packages/`

TaskViews: [LINK](http://star-www.st-andrews.ac.uk/cran/web/views/) `http://star-www.st-andrews.ac.uk/cran/web/views/`

Bioconductor is more focused on 'omics, and has a different curation mechanism

Here are the basics for installing stuff.

```
# -----
# Installing packages
# -----

# R-CRAN
```

```
# get gplots, dplyr, ggplot2
# see http://ggplot2.org and
# https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html

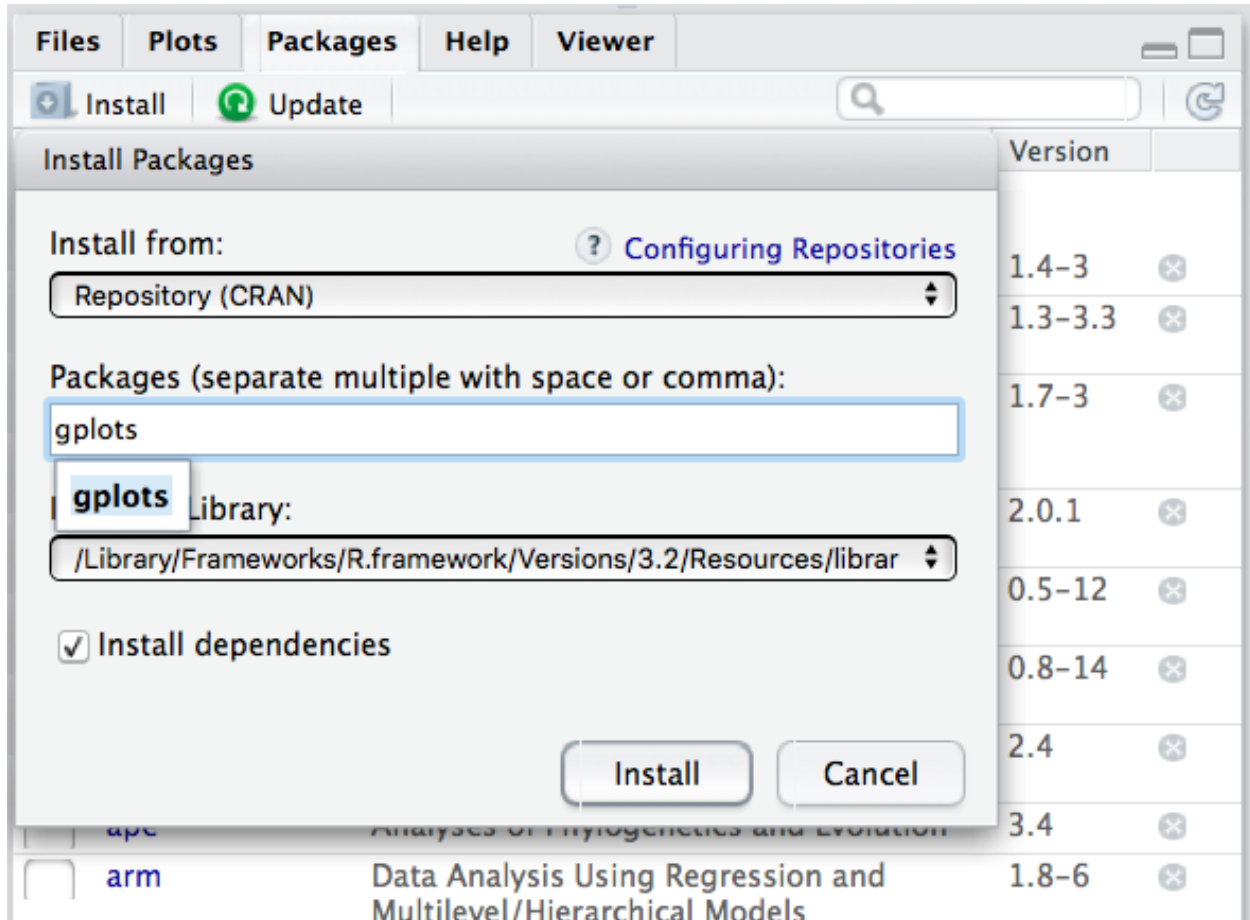
install.packages(c('gplots', 'ggplot2', 'dplyr'))

# R - bioconductor
# the first line grabs a special piece of code
# to make it easy to get packages from BIOCONDUCTOR.
# The second line gets a specific set of packages....
# you'll be using these!

source("https://bioconductor.org/biocLite.R")
biocLite(c("DESeq2", "KEGGREST", "pathview", "org.Dr.eg.db"))
```

Some stuff will happen on your screen when you do this...

Of course, you can also use RStudio to install packages.



Think about packages in this fun way

You should think about packages like this. You INSTALL PACKAGES from an app store. This is like downloading the app.

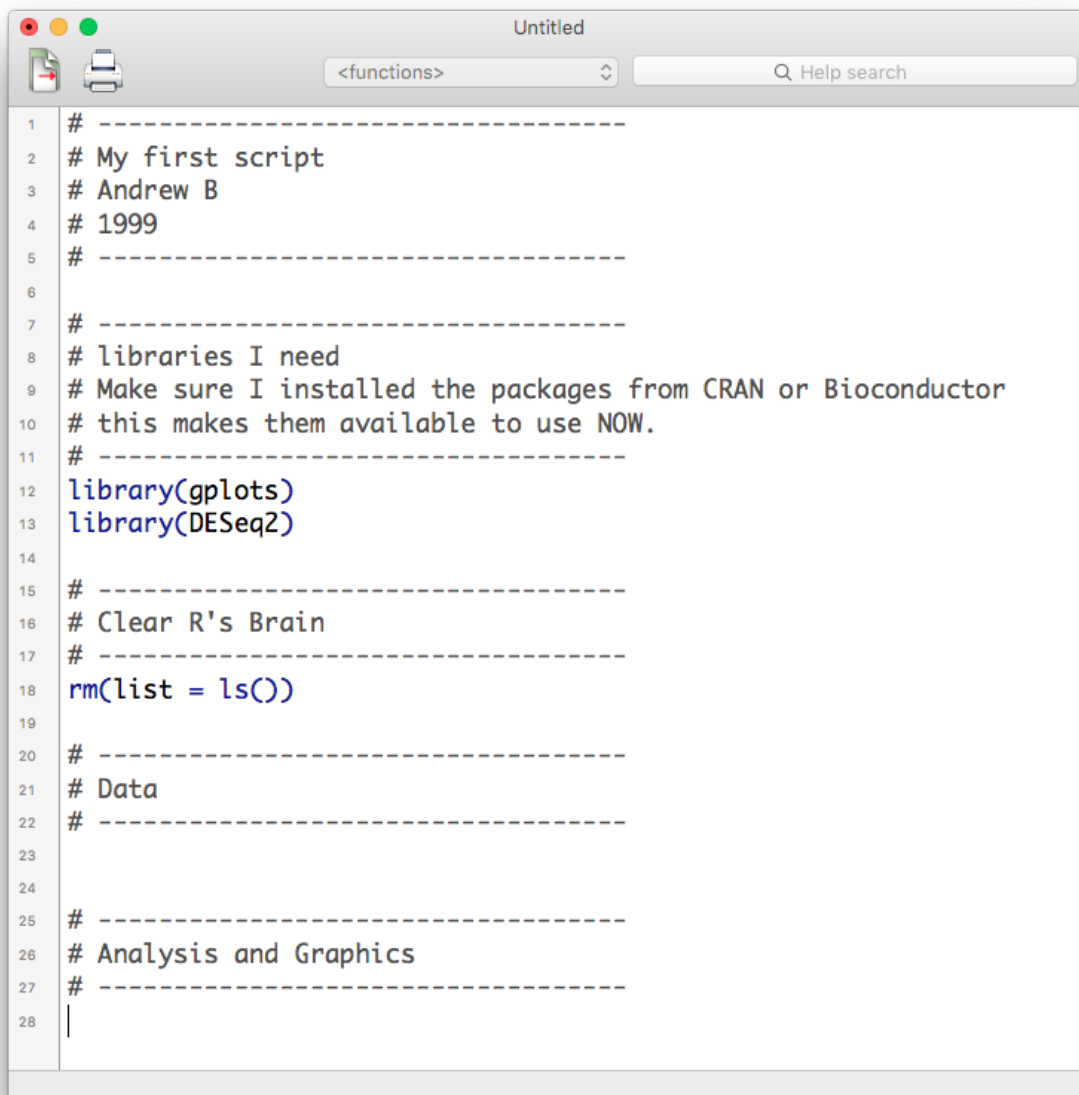
To use them, however, we need to invoke a second function (like pressing on the icon). This is called `library()`. It makes the package and all its Goodness executable.

Typically, we'd do the *intallation* which is a *download* from the app store in the CONSOLE or via RStudio. Then we need to put the `library()` commands at the top of your script... Remember we left a place holder for this?

```
# I need these libraries  
library(KEGGREST)  
library(pathview)  
library(org.Dr.eg.db)
```

This is what the beginning of your script might look like... good practice.

Try and make sure the start of your script is informative, has libraries you need and clears the deck before starting.



```
1 # -----
2 # My first script
3 # Andrew B
4 # 1999
5 # -----
6
7 # -----
8 # libraries I need
9 # Make sure I installed the packages from CRAN or Bioconductor
10 # this makes them available to use NOW.
11 # -----
12 library(gplots)
13 library(DESeq2)
14
15 # -----
16 # Clear R's Brain
17 # -----
18 rm(list = ls())
19
20 # -----
21 # Data
22 # -----
23
24
25 # -----
26 # Analysis and Graphics
27 # -----
28 |
```

PART II Starts here....

Reading data in from the web, or your own computer.

Add this to your script.

```
# -----
# read some data using read.csv()
# -----

# this is a path to data on the web
UrlAddress<-'https://raw.githubusercontent.com/andbeck/RDatas/master/myDF.csv'

# get the data
myDF <- read.csv(UrlAddress)
str(myDF)
```

```
## 'data.frame': 100 obs. of 4 variables:
## $ X : int 1 2 3 4 5 6 7 8 9 10 ...
## $ expression: num 12.52 1.31 9.02 9.5 10.06 ...
## $ size : num -0.506 1.8712 -0.1033 0.1141 0.0369 ...
## $ category : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
```

Don't add this...

```
# # with comp
myDF <- read.csv(compAddress)

# # also directly using a path
myDF <- read.csv('~/.Documents/Rstuff/file.csv')
```

Add this to your script...

```
# -----
# make a list (also common)
myList <- list(x = 1:10,
  y = LETTERS[1:5],
  w = matrix(rnorm(100,0,1), nrow = 10, ncol = 10, byrow = TRUE))
```

You will often have data on your computer, from machines etc.

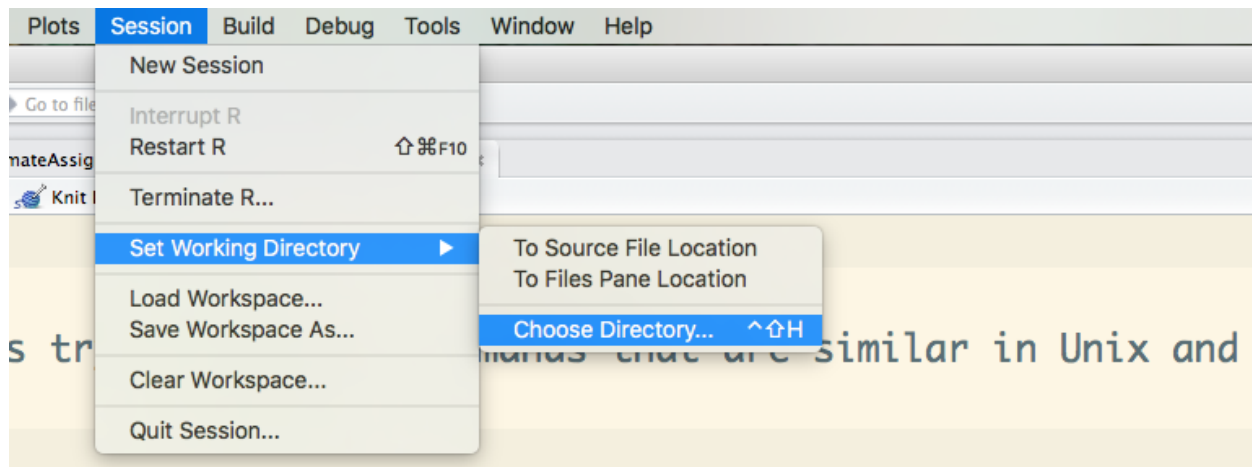
When you have your own data, you need to know where it is...

The path to the folder where the data is... You can use the path directly.

```
# # you will often use a path on your computer
## Windows and OSX/Linux use different slashes for paths
##... R no longer cares.

compAddress<-'C:\Documents\RStuff\myDF.csv' # windows
compAddress<-'~/Documents/RStuff/myDF.csv' # linux/unix/osx
```

Or you can set the working directory in RStudio



And then just provide the file name.

```
# After using RStudio to set the working directly
# just define the file

myDf<- read.csv('myDF.csv')
```

Finding your way around data.

Omics data structures are highly variable. Things are stored in data frames and in lists. Below you have - an introduction to `[row, column]`, `$` for grabbing bits of data frames - an introduction to `subset()` for more efficient grabbing - an introduction to `[[]]` for grabbing things from lists

You'll want to work through these... thoroughly.

-NOTE : lots of people are now using the package `dplyr` now with verbs `select()`, `slice()`, `filter()` and others as an alternative to BASE functions. There are VERY good tutorials online. These are very FAST with LARGE datasets.

```
# -----
# Explore the data with important functions in R
# [ ] and $ and ==
# REMEMBER: R works 99% of the time with ROWS THEN COLUMNS...
# e.g. [rows, columns]

# BASE BASICS
myDF[5,2] # 5th row, 2nd column
```

```
## [1] 10.05893
```

```
myDF$expression # use the $ to get the expression column
```

```
## [1] 12.517538 1.314097 9.016838 9.503552 10.058929 7.665571 10.888666
## [8] 7.978765 7.660154 8.647248 6.014360 13.729142 11.808561 6.199458
## [15] 5.672131 12.018454 15.260583 3.868344 14.726813 11.665405 12.494673
## [22] 15.193945 6.033581 6.425320 9.787794 15.281029 16.443706 3.192887
## [29] 13.885776 4.950545 3.195074 26.605668 12.156804 3.535982 12.737946
```

```
## [36] 21.454717 6.797553 15.073445 11.310967 11.433194 7.964187 6.546595
## [43] 14.422700 5.523245 14.150787 5.677373 22.440774 10.443891 20.035131
## [50] 9.031042 28.517471 29.475969 34.862050 30.531403 31.592105 32.048833
## [57] 31.461238 24.528072 38.931406 33.208995 30.994866 29.040634 27.535157
## [64] 27.531762 21.975596 26.610769 28.663481 26.534582 32.622669 34.101971
## [71] 28.728651 31.806852 30.507381 23.458605 33.575201 25.412200 35.492783
## [78] 33.556421 23.281725 30.938442 24.628242 23.968088 30.886866 33.312532
## [85] 23.507332 31.591445 25.820811 33.364705 21.973055 31.934126 31.137705
## [92] 27.894069 28.283657 29.103208 26.568325 35.225372 28.497635 37.298256
## [99] 35.073917 32.311718
```

```
myDF[,1] # ONLY 1st COLUMN
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
## [35] 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
## [52] 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
## [69] 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
## [86] 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

```
myDF[2,] # ONLY 2nd ROW
```

```
## X expression size category
## 2 2 1.314097 1.871247 A
```

```
myList[[3]] # double squares for lists; get the 3rd piece of the list => w
```

```
## [,1] [,2] [,3] [,4] [,5]
## [1,] 1.37477496 -2.10758858 0.707315268 1.4204070 1.13848384
## [2,] -0.05463575 0.17146010 0.216106636 0.3674432 -1.22301518
## [3,] 0.65240927 1.08364511 1.101012541 1.7714816 -1.20113169
## [4,] 0.94951429 -1.43990326 0.001274928 0.5869965 -2.44167518
## [5,] -0.13228937 -0.01828889 -0.027055975 -1.3108194 0.24243211
## [6,] 0.45046532 -0.95750574 -1.578028750 1.1359321 1.75366739
## [7,] 1.55339164 1.77419999 0.410560356 -0.2511082 -0.83488177
## [8,] 0.27778950 -0.59498696 -0.159636663 -1.4260941 -0.07614939
## [9,] 0.57867524 -2.31601968 0.851587103 -0.8340634 -0.57127205
## [10,] 0.80095974 1.89787525 1.358192742 -0.2532544 -1.27649564
## [,6] [,7] [,8] [,9] [,10]
## [1,] -0.4152332 1.15881561 0.94884269 -1.6200271 -0.03242861
## [2,] 0.9043322 -1.11090418 0.66253415 -1.3420178 0.70802776
## [3,] -0.3214752 0.72115322 0.23695201 -0.2207561 0.94504943
## [4,] 0.8693144 -0.46112812 -0.29978805 -0.2522404 0.52569272
## [5,] 0.8386452 -0.02736753 0.01134207 -0.3269844 -0.34383679
## [6,] 0.9073924 0.82526835 0.56574209 -0.6891510 -1.01254817
## [7,] 0.5000674 -1.46229296 0.41688635 -2.3185903 0.21970859
## [8,] 0.7071875 -1.41975302 0.12851135 0.5330683 -0.21950268
## [9,] 1.3016899 -0.89385100 0.73002787 0.4511705 -0.35082029
## [10,] 2.6590416 -0.21769536 0.42619710 0.2807444 1.90940537
```

```
# MORE harder
```

```
myList[["w"]][2,] # get the ninth row from the matrix w that is the third element in myList
```

```
## [1] -0.05463575 0.17146010 0.21610664 0.36744321 -1.22301518
```

```
## [6] 0.90433217 -1.11090418 0.66253415 -1.34201784 0.70802776
```

```
myList$w[2,] # get the ninth row from the matrix w that is the third element in myList
```

```
## [1] -0.05463575 0.17146010 0.21610664 0.36744321 -1.22301518
```

```
## [6] 0.90433217 -1.11090418 0.66253415 -1.34201784 0.70802776
```

```
# MORE Harder
```

```
myDF[51:100,] # rows 51:100 are labeled "B"
```

##		X	expression	size	category
## 51	51	28.51747	-0.45405598		B
## 52	52	29.47597	-1.97200873		B
## 53	53	34.86205	-0.50698562		B
## 54	54	30.53140	-0.61006478		B
## 55	55	31.59210	-0.57905778		B
## 56	56	32.04883	0.73580411		B
## 57	57	31.46124	1.62414287		B
## 58	58	24.52807	0.36799274		B
## 59	59	38.93141	-0.23602739		B
## 60	60	33.20899	1.36383951		B
## 61	61	30.99487	0.86499675		B
## 62	62	29.04063	0.30832280		B
## 63	63	27.53516	-0.83933275		B
## 64	64	27.53176	1.58860207		B
## 65	65	21.97560	1.16240904		B
## 66	66	26.61077	1.16098381		B
## 67	67	28.66348	-0.20333346		B
## 68	68	26.53458	0.92408774		B
## 69	69	32.62267	-1.32905326		B
## 70	70	34.10197	-0.06406315		B
## 71	71	28.72865	0.46624152		B
## 72	72	31.80685	0.84064708		B
## 73	73	30.50738	0.83803932		B
## 74	74	23.45861	0.76108249		B
## 75	75	33.57520	0.25438475		B
## 76	76	25.41220	0.84145960		B
## 77	77	35.49278	1.51905983		B
## 78	78	33.55642	-0.78395274		B
## 79	79	23.28172	0.06206436		B
## 80	80	30.93844	1.99305817		B
## 81	81	24.62824	0.89936350		B
## 82	82	23.96809	1.96480011		B
## 83	83	30.88687	0.89873445		B
## 84	84	33.31253	-0.01559538		B
## 85	85	23.50733	-0.18048792		B
## 86	86	31.59144	1.18288538		B
## 87	87	25.82081	-0.20844010		B

```
## 88 88 33.36471 -0.11618843 B
## 89 89 21.97306 1.92841533 B
## 90 90 31.93413 -2.01769093 B
## 91 91 31.13771 -0.29022348 B
## 92 92 27.89407 -0.14470734 B
## 93 93 28.28366 -0.10148508 B
## 94 94 29.10321 1.13478704 B
## 95 95 26.56832 1.68365002 B
## 96 96 35.22537 -0.42649623 B
## 97 97 28.49763 -1.86591499 B
## 98 98 37.29826 0.11377617 B
## 99 99 35.07392 0.53367742 B
## 100 100 32.31172 -1.33071604 B
```

```
myDF[myDF$category=="B",] # rows 51:100, but access functionally by definition in category column.
```

```
##      X expression      size category
## 51  28.51747 -0.45405598      B
## 52  29.47597 -1.97200873      B
## 53  34.86205 -0.50698562      B
## 54  30.53140 -0.61006478      B
## 55  31.59210 -0.57905778      B
## 56  32.04883 0.73580411      B
## 57  31.46124 1.62414287      B
## 58  24.52807 0.36799274      B
## 59  38.93141 -0.23602739      B
## 60  33.20899 1.36383951      B
## 61  30.99487 0.86499675      B
## 62  29.04063 0.30832280      B
## 63  27.53516 -0.83933275      B
## 64  27.53176 1.58860207      B
## 65  21.97560 1.16240904      B
## 66  26.61077 1.16098381      B
## 67  28.66348 -0.20333346      B
## 68  26.53458 0.92408774      B
## 69  32.62267 -1.32905326      B
## 70  34.10197 -0.06406315      B
## 71  28.72865 0.46624152      B
## 72  31.80685 0.84064708      B
## 73  30.50738 0.83803932      B
## 74  23.45861 0.76108249      B
## 75  33.57520 0.25438475      B
## 76  25.41220 0.84145960      B
## 77  35.49278 1.51905983      B
## 78  33.55642 -0.78395274      B
## 79  23.28172 0.06206436      B
## 80  30.93844 1.99305817      B
## 81  24.62824 0.89936350      B
## 82  23.96809 1.96480011      B
## 83  30.88687 0.89873445      B
## 84  33.31253 -0.01559538      B
## 85  23.50733 -0.18048792      B
## 86  31.59144 1.18288538      B
## 87  25.82081 -0.20844010      B
```

```
## 88 88 33.36471 -0.11618843 B
## 89 89 21.97306 1.92841533 B
## 90 90 31.93413 -2.01769093 B
## 91 91 31.13771 -0.29022348 B
## 92 92 27.89407 -0.14470734 B
## 93 93 28.28366 -0.10148508 B
## 94 94 29.10321 1.13478704 B
## 95 95 26.56832 1.68365002 B
## 96 96 35.22537 -0.42649623 B
## 97 97 28.49763 -1.86591499 B
## 98 98 37.29826 0.11377617 B
## 99 99 35.07392 0.53367742 B
## 100 100 32.31172 -1.33071604 B
```

```
# -----
# learn how to use subset (?subset, perhaps?)
# -----
subset(myDF, category=="B") # rows
```

```
##      X expression      size category
## 51 51 28.51747 -0.45405598 B
## 52 52 29.47597 -1.97200873 B
## 53 53 34.86205 -0.50698562 B
## 54 54 30.53140 -0.61006478 B
## 55 55 31.59210 -0.57905778 B
## 56 56 32.04883 0.73580411 B
## 57 57 31.46124 1.62414287 B
## 58 58 24.52807 0.36799274 B
## 59 59 38.93141 -0.23602739 B
## 60 60 33.20899 1.36383951 B
## 61 61 30.99487 0.86499675 B
## 62 62 29.04063 0.30832280 B
## 63 63 27.53516 -0.83933275 B
## 64 64 27.53176 1.58860207 B
## 65 65 21.97560 1.16240904 B
## 66 66 26.61077 1.16098381 B
## 67 67 28.66348 -0.20333346 B
## 68 68 26.53458 0.92408774 B
## 69 69 32.62267 -1.32905326 B
## 70 70 34.10197 -0.06406315 B
## 71 71 28.72865 0.46624152 B
## 72 72 31.80685 0.84064708 B
## 73 73 30.50738 0.83803932 B
## 74 74 23.45861 0.76108249 B
## 75 75 33.57520 0.25438475 B
## 76 76 25.41220 0.84145960 B
## 77 77 35.49278 1.51905983 B
## 78 78 33.55642 -0.78395274 B
## 79 79 23.28172 0.06206436 B
## 80 80 30.93844 1.99305817 B
## 81 81 24.62824 0.89936350 B
## 82 82 23.96809 1.96480011 B
## 83 83 30.88687 0.89873445 B
## 84 84 33.31253 -0.01559538 B
```

```
## 85 85 23.50733 -0.18048792 B
## 86 86 31.59144 1.18288538 B
## 87 87 25.82081 -0.20844010 B
## 88 88 33.36471 -0.11618843 B
## 89 89 21.97306 1.92841533 B
## 90 90 31.93413 -2.01769093 B
## 91 91 31.13771 -0.29022348 B
## 92 92 27.89407 -0.14470734 B
## 93 93 28.28366 -0.10148508 B
## 94 94 29.10321 1.13478704 B
## 95 95 26.56832 1.68365002 B
## 96 96 35.22537 -0.42649623 B
## 97 97 28.49763 -1.86591499 B
## 98 98 37.29826 0.11377617 B
## 99 99 35.07392 0.53367742 B
## 100 100 32.31172 -1.33071604 B
```

```
subset(myDF, select = "expression") # columns
```

```
## expression
## 1 12.517538
## 2 1.314097
## 3 9.016838
## 4 9.503552
## 5 10.058929
## 6 7.665571
## 7 10.888666
## 8 7.978765
## 9 7.660154
## 10 8.647248
## 11 6.014360
## 12 13.729142
## 13 11.808561
## 14 6.199458
## 15 5.672131
## 16 12.018454
## 17 15.260583
## 18 3.868344
## 19 14.726813
## 20 11.665405
## 21 12.494673
## 22 15.193945
## 23 6.033581
## 24 6.425320
## 25 9.787794
## 26 15.281029
## 27 16.443706
## 28 3.192887
## 29 13.885776
## 30 4.950545
## 31 3.195074
## 32 26.605668
## 33 12.156804
## 34 3.535982
```


## 35	12.737946
## 36	21.454717
## 37	6.797553
## 38	15.073445
## 39	11.310967
## 40	11.433194
## 41	7.964187
## 42	6.546595
## 43	14.422700
## 44	5.523245
## 45	14.150787
## 46	5.677373
## 47	22.440774
## 48	10.443891
## 49	20.035131
## 50	9.031042
## 51	28.517471
## 52	29.475969
## 53	34.862050
## 54	30.531403
## 55	31.592105
## 56	32.048833
## 57	31.461238
## 58	24.528072
## 59	38.931406
## 60	33.208995
## 61	30.994866
## 62	29.040634
## 63	27.535157
## 64	27.531762
## 65	21.975596
## 66	26.610769
## 67	28.663481
## 68	26.534582
## 69	32.622669
## 70	34.101971
## 71	28.728651
## 72	31.806852
## 73	30.507381
## 74	23.458605
## 75	33.575201
## 76	25.412200
## 77	35.492783
## 78	33.556421
## 79	23.281725
## 80	30.938442
## 81	24.628242
## 82	23.968088
## 83	30.886866
## 84	33.312532
## 85	23.507332
## 86	31.591445
## 87	25.820811
## 88	33.364705

```
## 89 21.973055
## 90 31.934126
## 91 31.137705
## 92 27.894069
## 93 28.283657
## 94 29.103208
## 95 26.568325
## 96 35.225372
## 97 28.497635
## 98 37.298256
## 99 35.073917
## 100 32.311718
```

```
subset(myDF, category=="B", select = "expression") # both
```

```
##      expression
## 51 28.51747
## 52 29.47597
## 53 34.86205
## 54 30.53140
## 55 31.59210
## 56 32.04883
## 57 31.46124
## 58 24.52807
## 59 38.93141
## 60 33.20899
## 61 30.99487
## 62 29.04063
## 63 27.53516
## 64 27.53176
## 65 21.97560
## 66 26.61077
## 67 28.66348
## 68 26.53458
## 69 32.62267
## 70 34.10197
## 71 28.72865
## 72 31.80685
## 73 30.50738
## 74 23.45861
## 75 33.57520
## 76 25.41220
## 77 35.49278
## 78 33.55642
## 79 23.28172
## 80 30.93844
## 81 24.62824
## 82 23.96809
## 83 30.88687
## 84 33.31253
## 85 23.50733
## 86 31.59144
## 87 25.82081
## 88 33.36471
```

```
## 89    21.97306
## 90    31.93413
## 91    31.13771
## 92    27.89407
## 93    28.28366
## 94    29.10321
## 95    26.56832
## 96    35.22537
## 97    28.49763
## 98    37.29826
## 99    35.07392
## 100   32.31172
```

summarising data

Here we introduce the use of BASE functions `aggregate` and `tapply`, which are used to generate means, standard deviations etc of data by grouping variables (e.g. parts of experimental designs)

-NOTE - lots of people are using the package `dplyr` now with verbs `select()`, `slice()`, `filter()` along with `summarise()` and `group_by()`; VERY good tutorials online

```
# -----
# Summarize the data using tapply and aggregate
# -----
# aggregate, formulae style.
aggregate(expression ~ category, data = myDF, FUN = mean)
```

```
##   category expression
## 1      A    10.60882
## 2      B    29.79817
```

```
# OR argument style; note that with() declares WHERE to look
# for the column names...

with(myDF,
      aggregate(x = expression, by = list(category), FUN = mean))
```

```
##   Group.1      x
## 1      A 10.60882
## 2      B 29.79817
```

```
# tapply
with(myDF,
      tapply(X = expression, INDEX = list(category), FUN = mean))
```

```
##      A      B
## 10.60882 29.79817
```

```
# YOU CAN assign the values returned by subset, aggregate and tapply to objects
mean.numbers <- with(myDF,
                     tapply(X = expression, INDEX = list(category), FUN = mean))
```

```
# show me
mean.nums
```

```
##           A           B
## 10.60882 29.79817
```

The apply family and loops

The **apply** family is a very powerful family of functions for working with data frames and “applying” functions to rows and columns. You see above that **tapply()** takes a column of a data frame, divides it up by the levels of a factor from another column, and applies a function to each group. VERY handy.

The function **apply()** allows us to “apply” a function to each row or column. This can be very useful. The first argument is the data frame, the second an indicator (1 or 2) of whether to work on rows or columns, and the third the function to use.

Lets imagine we want to divide each numeric variable in the small **myDat** data frame by 10. This is what we’d do.

```
apply(myDat, 2, function(x) {x/10})
```

```
##           Beer Goggles
## [1,]  0.1         0.1
## [2,]  0.2         0.4
## [3,]  0.3         0.9
## [4,]  0.4         1.6
## [5,]  0.5         2.5
## [6,]  0.6         3.6
## [7,]  0.7         4.9
## [8,]  0.8         6.4
## [9,]  0.9         8.1
## [10,] 1.0        10.0
```

Lets dissect this. We are working with the **myDat** data frame. We are going to work on the **COLUMNS**, as we’ve specified 2. The function we are going to use is **x/10**. This is interpreted as for every column **x**, do **x/10** for every element in **x**.

Don’t get confused by **x**. Because we have a 2, **x** is EACH column.

Transitioning to Unix.

Lets move focus to the console for a moment... wait. Have you saved the script? Use CTRL - 2. Ohhh - check out the View Menu... Once in the consol, type:

```
ls()
```

```
## [1] "mean.nums" "myDat"      "myDF"       "myList"     "UrlAddress"
## [6] "x"         "y"
```

Now, lets try a few other commands that are similar in Unix and one extra one.

```
head(myDF) # first six rows
```

```
##      X expression      size category
## 1 1  12.517538 -0.50601080          A
## 2 2   1.314097  1.87124678          A
## 3 3   9.016838 -0.10334363          A
## 4 4   9.503552  0.11412309          A
## 5 5  10.058929  0.03690971          A
## 6 6   7.665571  0.81555607          A
```

```
tail(myDF) # last six
```

```
##      X expression      size category
## 95  95  26.56832  1.6836500          B
## 96  96  35.22537 -0.4264962          B
## 97  97  28.49763 -1.8659150          B
## 98  98  37.29826  0.1137762          B
## 99  99  35.07392  0.5336774          B
## 100 100 32.31172 -1.3307160          B
```

```
str(myDF) # STRUCTURE
```

```
## 'data.frame':   100 obs. of  4 variables:
## $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
## $ expression: num  12.52 1.31 9.02 9.5 10.06 ...
## $ size        : num  -0.506 1.8712 -0.1033 0.1141 0.0369 ...
## $ category    : Factor w/ 2 levels "A","B": 1 1 1 1 1 1 1 1 1 1 ...
```

```
getwd() # get the working directory
```

```
## [1] "/Users/apb/Documents/Repos/TeachStuff/MDIBL"
```

```
list.files() # list the files in the working directory
```

```
## [1] "Birmingham_2016_MDIBL.R" "Birmingham_MDIBL_files"
## [3] "Birmingham_MDIBL.html"  "Birmingham_MDIBL.pdf"
## [5] "Birmingham_MDIBL.Rmd"   "figures"
## [7] "UnixIntro.html"         "UnixIntro.pdf"
## [9] "UnixIntro.Rmd"
```

```
# don't forget.... you can also use the menus via Session -> Set Working Directory to SET the working d
```

Loops.

You will use loops. Loops count for you and do things many times. Lets look at a simple loop:

```
for(i in 1:10) {print(i)}
```

This reads, count from 1 to 10, calling the counter i. At each i, print the number that is i.

Lets look at something more challenging

```
# Set each array value its native level plus noise
for(i in 1:10){
  array[,i] <- ControlExp + rnorm(1000, sd=.5)
}
```

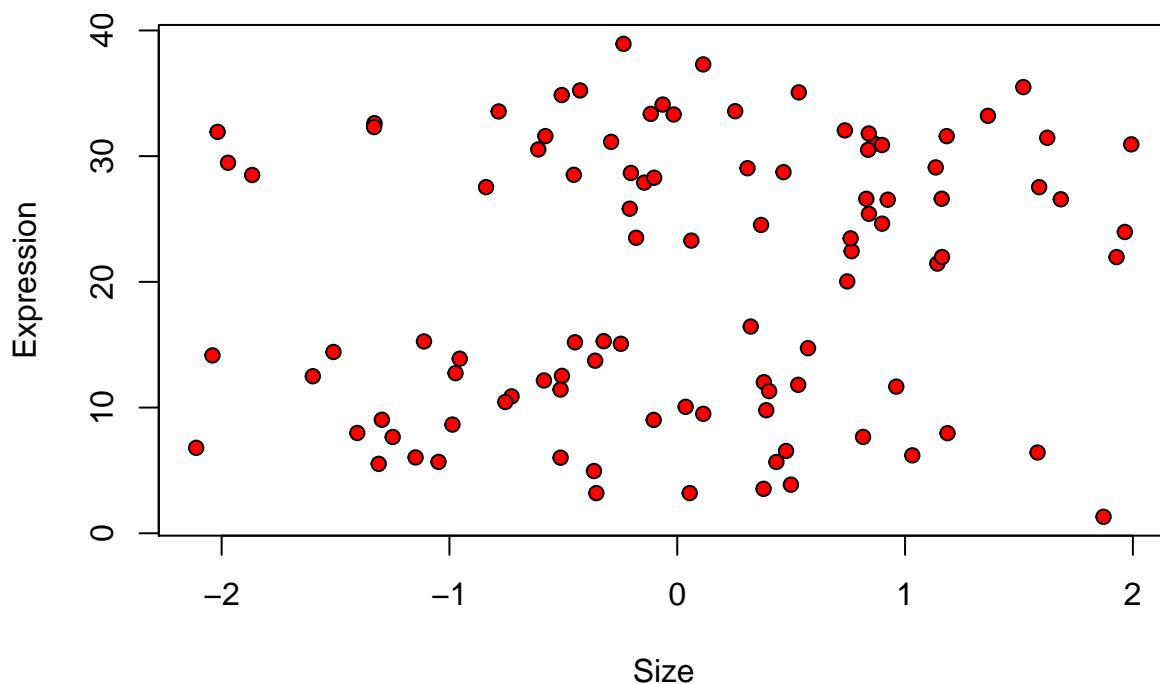
This has the same counter, but is now adding random numbers (`rnorm()`) to something called *ControlExp*, AND for each *i*, assigning the sum to the *i*'th column in *array* (which we know is by column because [row,column]).

a more in depth overview of various plottings you can do

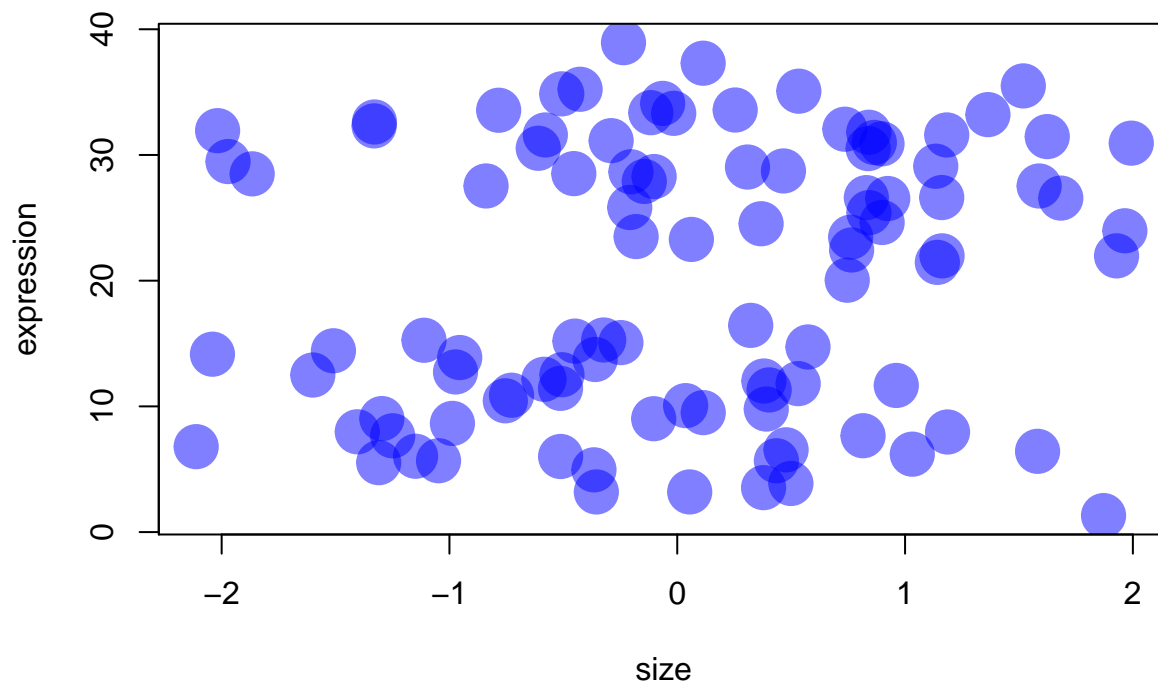
MANY methods for making plots with omics data. - advanced use of `plot()`- introduction to `ggplot2`- introduction to `lattice`

```
# -----
# make some different types of plots
# plot, barplot, histograms, heatmaps
# reinforce function and arguments
# -----

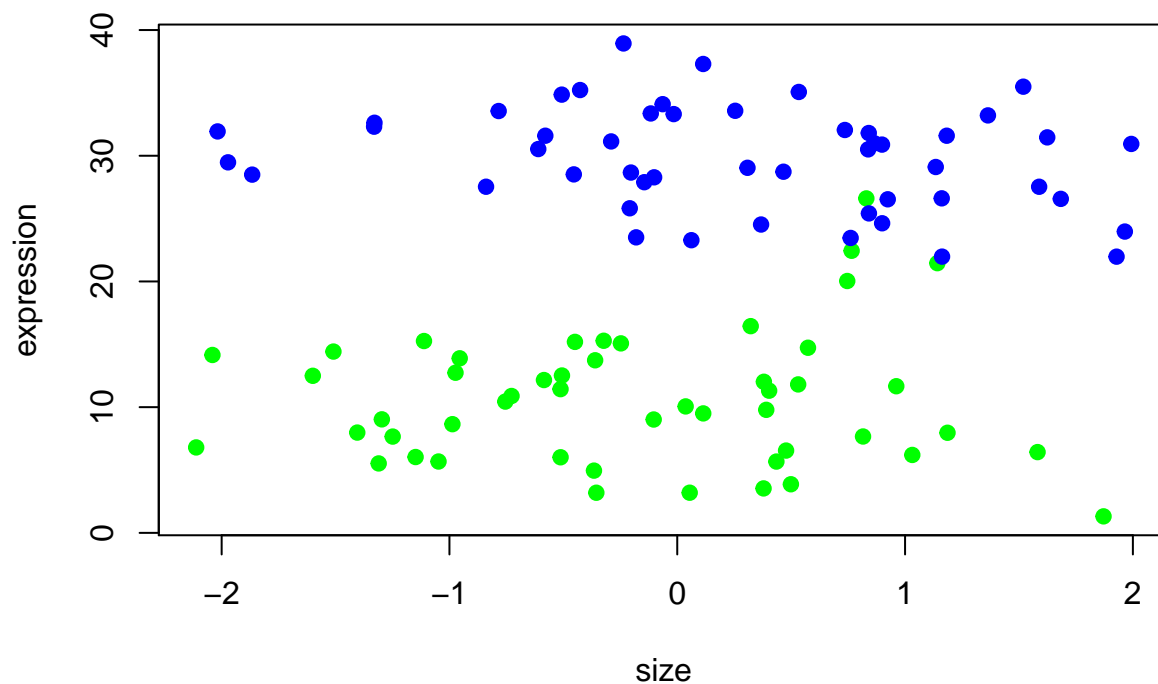
# plot, using the formula method - the best
plot(expression ~ size, data = myDF, pch = 21, bg = "red",
      xlab="Size", ylab = "Expression")
```



```
# alter the color, and use alpha transparency
plot(expression ~ size, data = myDF, pch = 19, cex = 3, col = rgb(0,0,1,alpha = 0.5))
```

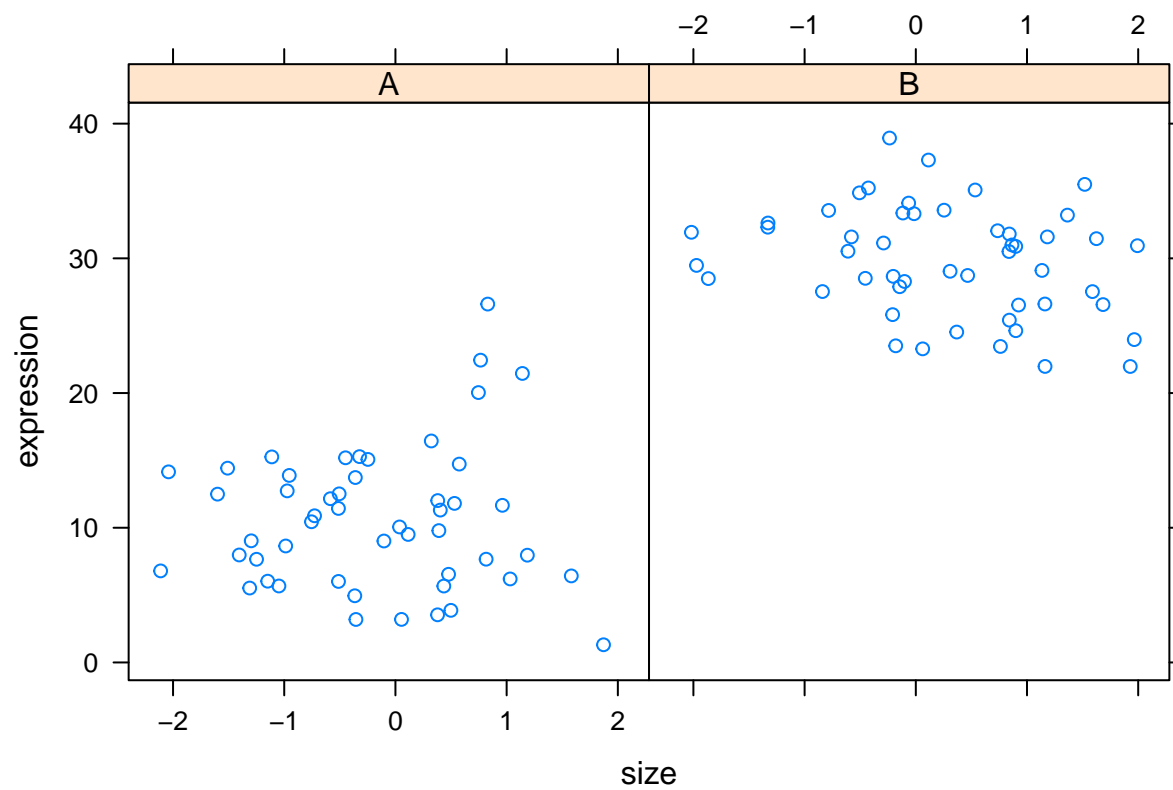


```
# specifying categories of colours on the points based on a categorical column
# note the special use of []'s
plot(expression ~ size, data = myDF,
      pch = 19, col = c("green", "blue")[category])
```

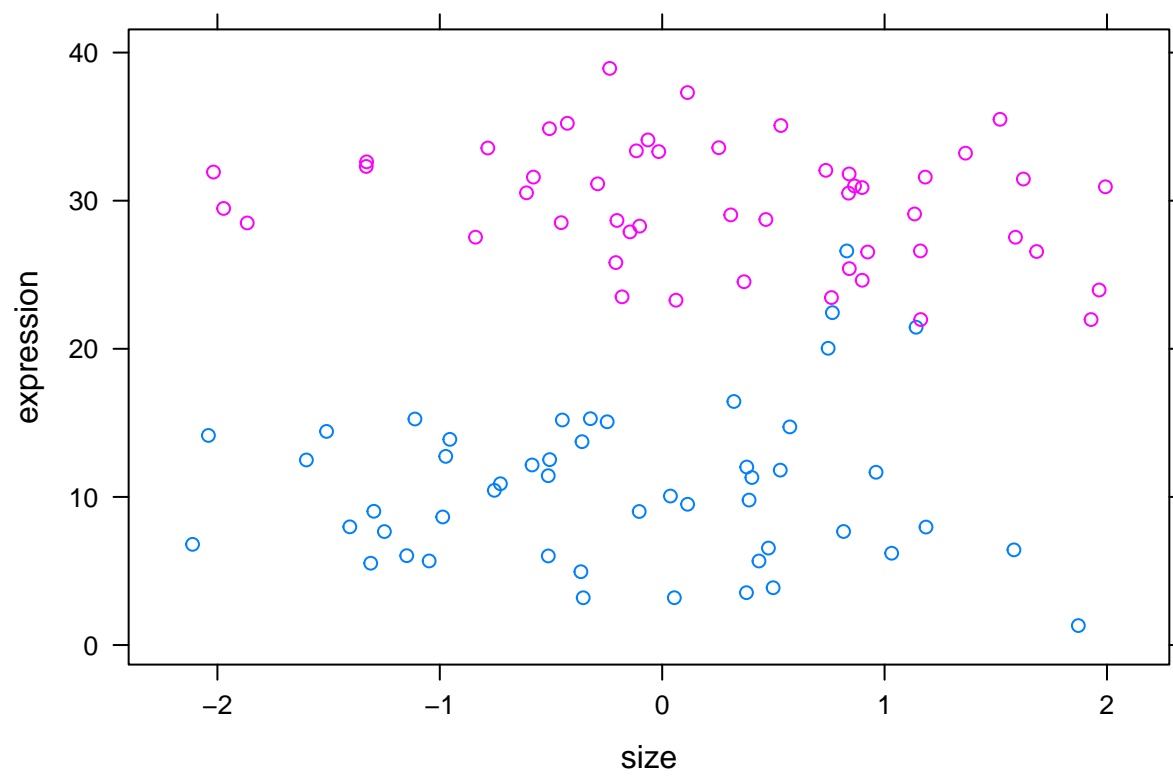


```
# lattice graphics - super fun exploration
library(lattice) # loads a special graphics library into R's brain

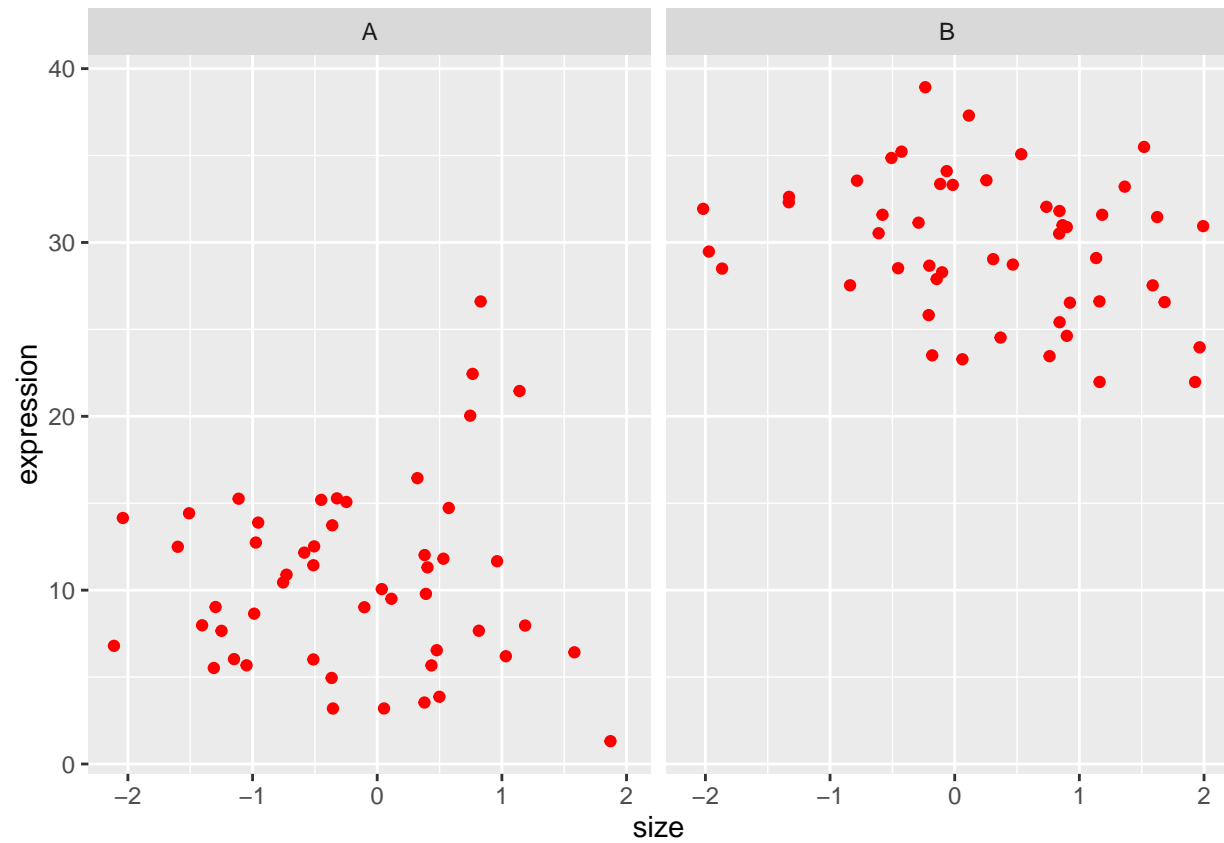
xyplot(expression ~ size | category, data = myDF) # panels
```



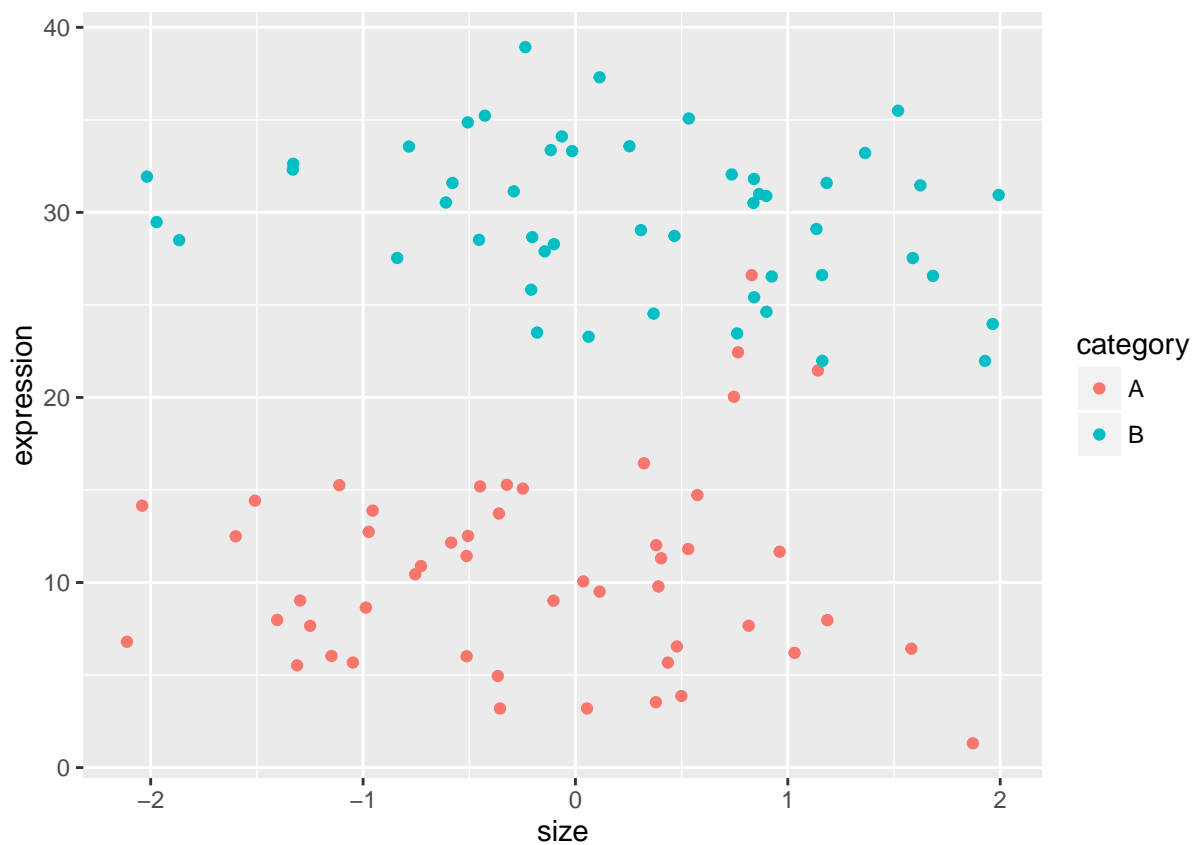
```
xyplot(expression ~ size, groups = category, data = myDF) # groups
```




```
# ggplot graphics - very popular now too
library(ggplot2)
ggplot(myDF, aes(x = size, y = expression))+
  geom_point(col = "red")+
  facet_wrap(~category)
```



```
ggplot(myDF, aes(x = size, y = expression, col = category))+
  geom_point()
```



```
# heatmaps - very common in molecular data
library(gplots)
```

```
##
## Attaching package: 'gplots'

## The following object is masked from 'package:stats':
##
## lowess
```

```
heatmap.2(myList$w,
          dendrogram="both",
          trace="none")
```

