Project Phoenix learnR Session 1 UNAM

Dr Mark Kelson - University of Exeter

12 June 2017

Figure 1:

Welcome to learnR

This course will teach you the basics of the R programming language and environment. By the end of the course you will be adept at:

Importing/exporting data

Graphically exploring your data

Manipulating your data

Analysing your data

Interpreting the results of your analysis

Reporting your analyses

Throughout this course we will take a dataset as our fundamental object from which all of our examples will start. This hopefully mirrors the real-life situation that you find yourself in: having a dataset and needing to analyse it.

Importing data

Data can be imported in R in a variety of ways. Type

```
?read.table
```

into the console window. A help file will open describing the read.table function. We will now use the read.csv function to import a csv file. Locate the Delegates.csv file on your USB stick and note it's location. Then type the following

```
my.data <- read.csv(file="***INSERT THE REST OF THE FILE LOCATION HERE***//Delegates.csv")
```

The read.table function, reads the csv file at the specified location. The assignment operator is " <-" indicates that the data file is to be called "my.data". We can examine the data by typing it's name into the console window. Try it and see what happens.

```
my.data
```

This is not the best way to view the data! Instead we might want to know the variable names. Then we would type

```
col.names(my.data)
```

Alternatively we might want to view just the top of the file. Then we could type

```
head(my.data)
```

The function "read.table" is a more generic function and can be used to read other file types. We can also use the argument "sep" to indicate what character is used to separate data points in the file (in csv files this is a comma). We can see what is in our data file by typing

```
ls()
```

Packages in R are suites of functions that people write and make available online. There are various packages that contain functions for importing other file types, including the "foreign" package for importing SPSS files. We will now download and use this package. To download a package we can either use the dropdown menu or syntax. First, the dropdown menu. Click on "Tools", then "Install Packages". Type in "foreign" into the middle box.

You will see that this action has pasted the following code into your console window

```
> install.packages("foreign")
```

Now that we have downloaded the library, we need to attach it to our current session by typing

```
library(foreign)
```

We can import the SPSS file XXXXX and save it in our current workspace as "spss.data" by noting it's location and typing it in as follows

```
spss.data <- read.spss(file="",to.data.frame=T, use.value.labels=T)
```

We can look at this data using the "head" command as before. Notice how the labels from SPSS have been transferred.

More support on importing file types is available hereand here

Data can also be exported using the write.csv command. Try saving the SPSS data as a csv file in the same location. If you need help you can type

```
?write.csv
```

Manipulating data

Once a dataset has been imported we can start interrogating it. If we know the variable name we can examine it on it's own. Say we want to look at the number variable in the delegates data. We would type

```
my.data$number
```

We can get summary information on this variable using the summary command. Try it now (remember, you can type "?summary" for help). Write in the mean and standard deviation in your workbook below.

Mean: _____

Standard deviation: _____

Good data management practice ensures that variables are in columns and observations in rows. Sometimes however, data isn't always presented in the right format. Say, we had a dataset where vaariables were in rows and observations in columns. We can construct a toy dataset easily. Type

```
wrongway <- cbind(A= c(1,2,3),B=c(4,5,6),C=c(7,8,9))
```

We can swap rows for columns with the transpose command "t"

```
rightway <- t(wrongway)
```

Now check that this has worked. Sometimes variable names are too long or unwieldy and may need to be changed. This can done using the "colnames" command. Try naming the columns in "rightway", Var1,Var2 and Var3 using the colnames command (remember use ?colnames for help).

Aggregating data

This is taken from here https://www.r-statistics.com/2012/01/aggregation-and-restructuring-data-from-r-in-action/

It's relatively easy to collapse data in R using one or more by variables and a defined function. The format is

```
aggregate(x, by, FUN)
```

where x is the data object to be collapsed, by is a list of variables that will be crossed to form the new observations, and FUN is the scalar function used to calculate summary statistics that will make up the new observation values.

As an example, we'll aggregate the mtcars data by number of cylinders and gears, returning means on each of the numeric variables (see the next listing).

Listing 2 aggregating data

```
options(digits=3)
attach(mtcars)
aggdata <-aggregate(mtcars, by=list(cyl,gear), FUN=mean, na.rm=TRUE)
aggdata
```

```
##   Group.1 Group.2  mpg cyl disp  hp drat   wt qsec  vs   am gear carb
## 1       4       3 21.5   4  120  97 3.70 2.46 20.0 1.0 0.00    3 1.00
## 2       6       3 19.8   6  242 108 2.92 3.34 19.8 1.0 0.00    3 1.00
## 3       8       3 15.1   8  358 194 3.12 4.10 17.1 0.0 0.00    3 3.08
## 4       4       4 26.9   4  103  76 4.11 2.38 19.6 1.0 0.75    4 1.50
## 5       6       4 19.8   6  164 116 3.91 3.09 17.7 0.5 0.50    4 4.00
## 6       4       5 28.2   4  108 102 4.10 1.83 16.8 0.5 1.00    5 2.00
## 7       6       5 19.7   6  145 175 3.62 2.77 15.5 0.0 1.00    5 6.00
## 8       8       5 15.4   8  326 300 3.88 3.37 14.6 0.0 1.00    5 6.00
```

In these results, Group.1 represents the number of cylinders (4, 6, or 8) and Group.2 represents the number of gears (3, 4, or 5). For example, cars with 4 cylinders and 3 gears have a mean of 21.5 miles per gallon (mpg).

When you're using the aggregate() function , the by variables must be in a list (even if there's only one). You can declare a custom name for the groups from within the list, for instance, using by=list(Group.cyl=cyl, Group.gears=gear).

The function specified can be any built-in or user-provided function. This gives the aggregate command a great deal of power. But when it comes to power, nothing beats the reshape package.

The reshape package

The reshape package is a tremendously versatile approach to both restructuring and aggregating datasets. Because of this versatility, it can be a bit challenging to learn.

We'll go through the process slowly and use a small dataset so that it's clear what's happening. Because reshape isn't included in the standard installation of R, you'll need to install it one time, using install.packages("reshape").

Basically, you'll "melt" data so that each row is a unique ID-variable combination. Then you'll "cast" the melted data into any shape you desire. During the cast, you can aggregate the data with any function you wish. The dataset you'll be working with is shown in table 1.

Table 1 The original dataset (mydata)

ID	Time	X1	X2
1	1	5	6
1	2	3	5
2	1	6	1
2	2	2	4

Create this dataset by typing

```
mydata <- data.frame(ID=c(1,1,2,2),Time=c(1,2,1,2),X1=c(5,3,6,2),X2=c(6,5,1,4))
```

In this dataset, the measurements are the values in the last two columns (5, 6, 3, 5, 6, 1, 2, and 4). Each measurement is uniquely identified by a combination of ID variables (in this case ID, Time, and whether the measurement is on X1 or X2). For example, the measured value 5 in the first row is uniquely identified by knowing that it's from observation (ID) 1, at Time 1, and on variable X1.

##Melting

When you melt a dataset, you restructure it into a format where each measured variable is in its own row, along with the ID variables needed to uniquely identify it. If you melt the data from table 1, using the following code

```
library(reshape)
md <- melt(mydata, id=(c("id", "time")))
```

You end up with the structure shown in table 2.

Table 2 The melted dataset

ID	Time	Variable	Value
1	1	X1	5
1	2	X1	3
2	1	X1	6
2	2	X1	2
1	1	X2	6
1	2	X2	5
2	1	X2	1
2	2	X2	4

Note that you must specify the variables needed to uniquely identify each measurement (ID and Time) and that the variable indicating the measurement variable names (X1 or X2) is created for you automatically.

Now that you have your data in a melted form, you can recast it into any shape, using the cast() function.

Casting

The cast() function starts with melted data and reshapes it using a formula that you provide and an (optional) function used to aggregate the data. The format is

```
newdata <- cast(md, formula, FUN)
```

Where md is the melted data, formula describes the desired end result, and FUN is the (optional) aggregating function. The formula takes the form

```
rowvar1 + rowvar2 + .  ~  colvar1 + colvar2 + .
```

In this formula, rowvar1 + rowvar2 + . define the set of crossed variables that define the rows, and colvar1 + colvar2 + . define the set of crossed variables that define the columns. See the examples in figure 2.

Because the formulas on the right side (d, e, and f) don't include a function, the data is reshaped. In contrast, the examples on the left side (a, b, and c) specify the mean as an aggregating function. Thus the data are not only reshaped but aggregated as well. For example, (a) gives the means on X1 and X2 averaged over time for each observation. Example (b) gives the mean scores of X1 and X2 at Time 1 and Time 2, averaged over observations. In (c) you have the mean score for each observation at Time 1 and Time 2, averaged over X1 and X2.

As you can see, the flexibility provided by the melt() and cast() functions is amazing. There are many times when you'll have to reshape or aggregate your data prior to analysis. For example, you'll typically need to place your data in what's called long format resembling table 2 when analyzing repeated measures data (data where multiple measures are recorded for each observation).

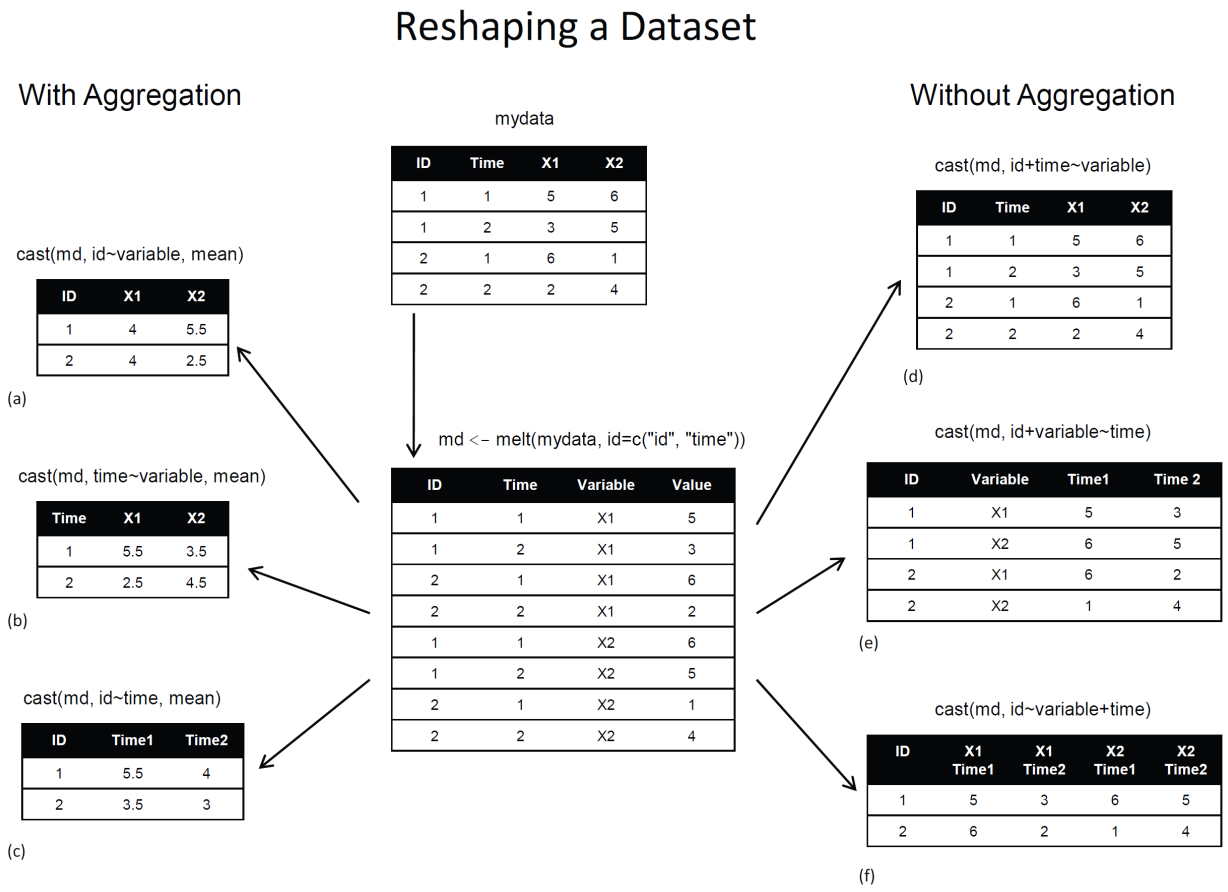

Figure 2: Reshaping data with the melt() and cast() functions

Data.Table

This material is taken from here https://www.dezyre.com/data-science-in-r-programming-tutorial/r-data-table-tutorial

Data.table is an extension of data.frame package in R. It is widely used for fast aggregation of large datasets, low latency add/update/remove of columns, quicker ordered joins, and a fast file reader. The syntax for data.table is flexible and intuitive and therefore leads to faster development. Some of the other notable features of data.tables are its fast primary ordered indexing and its automatic secondary indexing, this is complemented by a memory efficient combined join and group by. First we load the data.table PACKAGE by typing MAY NEED TO COVER LOADING PACKAGES FIRST

```
library(data.table)
```

The syntax for using a data.table is mentioned below:

```
DT[where, select|update|do, by]
```

Creating Data.Table

If you have created a data.frame before, you could recall that it is done by using the function data.frame():

```
DF = data.frame(x=c("b","b","b","a","a"),v=rnorm(5))
DF
```

```
##   x      v
## 1 b  0.334
## 2 b -0.177
## 3 b -0.416
## 4 a  0.372
## 5 a -0.344
```

A data.table is created in exactly the same way:

```
DT = data.table(x=c("b","b","b","a","a"),v=rnorm(5))
DT
```

```
##    x      v
## 1: b  0.352
## 2: b -0.157
## 3: b -0.482
## 4: a  0.117
## 5: a  0.873
```

Observe that a data.table prints the row numbers with a colon so as to visually separate the row number from the first column. We can easily convert existing data.frame objects to data.table.

```
MOTOR = data.table(mtcars)
head(MOTOR)
```

```
##     mpg cyl disp  hp drat   wt qsec vs am gear carb
## 1: 21.0   6  160 110 3.90 2.62 16.5  0  1    4    4
## 2: 21.0   6  160 110 3.90 2.88 17.0  0  1    4    4
## 3: 22.8   4  108  93 3.85 2.32 18.6  1  1    4    1
## 4: 21.4   6  258 110 3.08 3.21 19.4  1  0    3    1
## 5: 18.7   8  360 175 3.15 3.44 17.0  0  0    3    2
## 6: 18.1   6  225 105 2.76 3.46 20.2  1  0    3    1
```

We have just created two data.tables: DT and MOTORS. It is often useful to see a list of all data.tables in memory:

```
tables()
```

```
##      NAME  NROW NCOL MB COLS                                                KEY
## [1,] DT       5    2  1 x,v
## [2,] MOTOR   32   11  1 mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
## Total: 2MB
```

The MB column is useful to quickly assess memory use and to spot if any redundant tables can be removed to free up memory. Just like data.frames, data.tables must fit inside RAM. Some users regularly work with 20 or more tables in memory, rather like a database. The result of tables() is itself a data.table, returned silently, so that tables() can be used in programs. Tables() is unrelated to the base function table(). To see the column types :

```
sapply(DT,class)
```

```
##           x           v
## "character"   "numeric"
```

You may have noticed the empty column KEY in the result of tables() above. ##Keys Let's start by considering data.frame, specifically rownames. We know that each row has exactly one row name. However, a person (for example) has at least two names, a first name and a second name. It's useful to organise a telephone directory sorted by surname then first name. In data.table, a key consists of one or more columns. These columns may be integer, factor or numeric as well as character. Furthermore, the rows are sorted by the key. Therefore, a data.table can have at most one key because it cannot be sorted in more than one way. We can think of a key as like super-charged row names; i.e., mult-column and multi-type.

Uniqueness is not enforced; i.e., duplicate key values are allowed. Since the rows are sorted by the key, any duplicates in the key will appear consecutively. Let's remind ourselves of our tables:

```
tables()
```

```
##      NAME  NROW NCOL MB COLS                                                KEY
## [1,] DT       5    2  1 x,v
## [2,] MOTOR   32   11  1 mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
## Total: 2MB
```

No keys have been set yet.

```
DT[2,]    # select row 2
```

```
##    x      v
## 1: b -0.157
```

```
DT[x=="a",]    # select rows where column x == "a"
```

```
##    x     v
## 1: a 0.117
## 2: a 0.873
```

Aside: notice that we did not need to prefix x with DT$x. In data.table queries, we can use column names as if they are variables directly. But since there are no rownames, the following does not work:

```
cat(try(DT["a",],silent=TRUE))
```

```
## Error in `[.data.table`(DT, "a", ) :
##   When i is a data.table (or character vector), the columns to join by must be specified either using
```

When i is a data.table (or character vector), x must be keyed (i.e. sorted, and, marked as sorte The error message tells us we need to use setkey():

```
setkey(DT,x)
DT
```

```
##    x      v
## 1: a  0.117
## 2: a  0.873
## 3: b  0.352
## 4: b -0.157
## 5: b -0.482
```

Notice that the rows in DT have now been re-ordered according to the values of x. The two "a" rows have moved to the top. We can confirm that DT does indeed have a key using haskey(), key(), attributes(), or just running tables().

```
tables()
```

```
##      NAME  NROW NCOL MB COLS                                        KEY
## [1,] DT       5    2  1 x,v                                         x
## [2,] MOTOR   32   11  1 mpg,cyl,disp,hp,drat,wt,qsec,vs,am,gear,carb
## Total: 2MB
```

Now that we are sure DT has a key, let's try again:

```
DT["a"]
```

```
##    x     v
## 1: a 0.117
## 2: a 0.873
```

By default all the rows in the group are returned. The mult argument (short for multiple) allows the first or last row of the group to be returned instead.

```
DT["a", mult="first"]
```

```
##    x     v
## 1: a 0.117
```

```
DT["a", mult="last"]
```

```
##    x     v
## 1: a 0.873
```

Note: The comma between the arguments is optional.

Let's now create a new data.frame. We will make it large enough to demonstrate the difference between a vector scan and a binary search.

```
grpsize = ceiling(1e7/26^2) # 10 million rows, 676 groups

tt=system.time( DF <- data.frame(
 x=rep(LETTERS,each=26*grpsize),
 y=rep(letters,each=grpsize),
 v=runif(grpsize*26^2),
 stringsAsFactors=FALSE)
)
head(DF,3)
```

```
##   x y      v
## 1 A a 0.1716
## 2 A a 0.0247
## 3 A a 0.8994
```

```
tail(DF,3)
```

```
##          x y     v
## 10000066 Z z 0.168
## 10000067 Z z 0.967
## 10000068 Z z 0.536
```

```
dim(DF)
```

```
## [1] 10000068        3
```

We might say that R has created a 3 column table and inserted 10,000,068 rows. It took 0.457 secs, so it inserted 21,881,986 rows per second. This is normal in base R. Notice that we set stringsAsFactors=FALSE. This makes it a little faster for a fairer comparison, but feel free to experiment. Let's extract an arbitrary group from DF:

```
tt=system.time(ans1 <- DF[DF$x=="R" & DF$y=="h",]) # 'vector scan'
 head(ans1,3)
```

```
##         x y     v
## 6642058 R h 0.928
## 6642059 R h 0.473
## 6642060 R h 0.134
```

```
dim(ans1)
```

```
## [1] 14793     3
```

Now convert to a data.table and extract the same group:

```
DT = as.data.table(DF)    # It is recommended to use fread() or data.table() directly
system.time(setkey(DT,x,y)) # one-off cost
```

```
##    user  system elapsed
##   0.063   0.001   0.065
```

```
ss=system.time(ans2 <- DT[list("R","h")]) # binary search
head(ans2,3)
```

```
##    x y     v
## 1: R h 0.928
## 2: R h 0.473
## 3: R h 0.134
```

```
 dim(ans2)
```

```
## [1] 14793     3
```

```
 identical(ans1$v, ans2$v)
```

```
## [1] TRUE
```

At 0.001 seconds, this was 544 times faster than 0.544 seconds, and produced precisely the same result. If you are thinking that a few seconds is not much to save, it's the relative speedup that's important. The vector scan is linear, but the binary search is O(log n). It scales. If a task taking 10 hours is sped up by 100 times to 6 minutes, that is significant. We can do vector scans in data.table, too.

```
 system.time(ans1 <- DT[x=="R" & y=="h",]) # Not so efficient use of data.table
```

```
##    user  system elapsed
##   0.151   0.002   0.153
```

```
 system.time(ans2 <- DF[DF$x=="R" & DF$y=="h",]) # the data.frame way
```

```
##    user  system elapsed
##   0.169   0.034   0.203
```

```
mapply(identical,ans1,ans2)
```

```
##    x    y    v
## TRUE TRUE TRUE
```

If the phone book analogy helped, the 544 times speedup should not be surprising. We use the key to take advantage of the fact that the table is sorted and use binary search to find the matching rows. When we used x=="R" we scanned the entire column x, testing each and every value to see if it equalled "R". We did it again in the y column, testing for "h". Then & combined the two logical results to create a single logical vector which was passed to the [method, which in turn searched it for TRUE and returned those rows. These were vectorized operations. They occurred internally in R and were very fast, but they were scans. We did those scans because we wrote that R code.

When i is a list (and data.table is a list too), we say that we are joining. In this case, we are joining DT to the 1 row, 2 column table returned by list("R","h"). Since we do this a lot, there is an alias for list: .().

```
identical( DT[list("R","h"),],DT[.("R","h"),])
```

```
## [1] TRUE
```

Both vector scanning and binary search are available in data.table, but one way of using data.table is much better than the other. The join syntax is a short, fast to write and easy to maintain. Passing a data.table into a data.table subset is analogous to A[B] syntax in base R where A is a matrix and B is a 2-column matrix. In fact, the A[B] syntax in base R inspired the data.table package. There are other types of ordered joins and further arguments which are beyond the scope of this quick introduction.

The merge method of data.table is very similar to X[Y], but there are some differences. This first section has been about the first argument inside DT[...], namely i. The next section is about the 2nd and 3rd arguments of data.table: j and by. #Fast grouping using j and by The second argument to DT[...] is j and may consist of one or more expressions whose arguments are (unquoted) column names, as if the column names were variables. Just as we saw earlier in i as well.

```
DT[,sum(v)]
```

```
## [1] 5e+06
```

The by in data.table is fast. Let's compare it to tapply.

```
ttt=system.time(tt <- tapply(DT$v,DT$x,sum)); ttt
```

```
##    user  system elapsed
##   0.571   0.138   0.711
```

```
sss=system.time(ss <- DT[,sum(v),by=x]); sss
```

```
##    user  system elapsed
##   0.189   0.013   0.203
```

```
head(tt)
```

```
##      A      B      C      D      E      F
## 192208 192506 192453 192435 192225 192437
```

```
head(ss)
```

```
##    x     V1
## 1: A 192208
## 2: B 192506
## 3: C 192453
## 4: D 192435
## 5: E 192225
## 6: F 192437
```

```
identical(as.vector(tt), ss$V1)
```

```
## [1] TRUE
```

At 0.078 sec, this was 10 times faster than 0.767 sec, and produced precisely the same result. Next, let's group by two columns:

```
ttt=system.time(tt <- tapply(DT$v,list(DT$x,DT$y),sum)); ttt
```

```
##    user  system elapsed
##   1.239   0.278   1.520
```

```
sss=system.time(ss <- DT[,sum(v),by="x,y"]); sss
```

```
##    user  system elapsed
##   0.240   0.002   0.243
```

```
tt[1:5,1:5]
```

```
##      a    b    c    d    e
## A 7479 7412 7339 7399 7457
## B 7327 7399 7431 7387 7429
## C 7419 7360 7354 7357 7414
## D 7353 7365 7314 7444 7373
## E 7368 7385 7363 7371 7390
```

```
head(ss)
```

```
##    x y   V1
## 1: A a 7479
## 2: A b 7412
## 3: A c 7339
## 4: A d 7399
## 5: A e 7457
## 6: A f 7373
```

```
identical(as.vector(t(tt)), ss$V1)
```

```
## [1] TRUE
```

This was 10 times faster, and the syntax is a little simpler and easier to read.

Fast ordered joins

This is also known as last observation carried forward (LOCF) or a rolling join.

Recall that X[Y] is a join between data.table X and data.table Y. If Y has 2 columns, the first column is matched to the first column of the key of X and the 2nd column to the 2nd. An equi-join is performed by default, meaning that the values must be equal. Instead of an equi-join, a rolling join is :

```
X[Y,roll=TRUE]
```

As before the first column of Y is matched to X where the values are equal. The last join column in Y though, the 2nd one in this example, is treated specially. If no match is found, then the row before is returned, provided the first column still matches. Further controls are rolling forwards, backwards, nearest and limited staleness.

Graphing

This is taken from here

Everyone: Download workshop materials:

Download materials from http://tutorials.iq.harvard.edu/R/Rgraphics.zip Extract the zip file containing the materials to your desktop Workshop notes are available in .hmtl format. Open a file browser, navigate to your desktop and open Rgraphics.html

Why ggplot2?

Advantages of ggplot2

- consistent underlying grammar of graphics (Wilkinson, 2005)
- plot specification at a high level of abstraction
- very flexible
- theme system for polishing plot appearance
- mature and complete graphics system
- many users, active mailing list

That said, there are some things you cannot (or should not) do With ggplot2:

- 3-dimensional graphics (see the rgl package)
- Graph-theory type graphs (nodes/edges layout; see the igraph package)
- Interactive graphics (see the ggvis package)

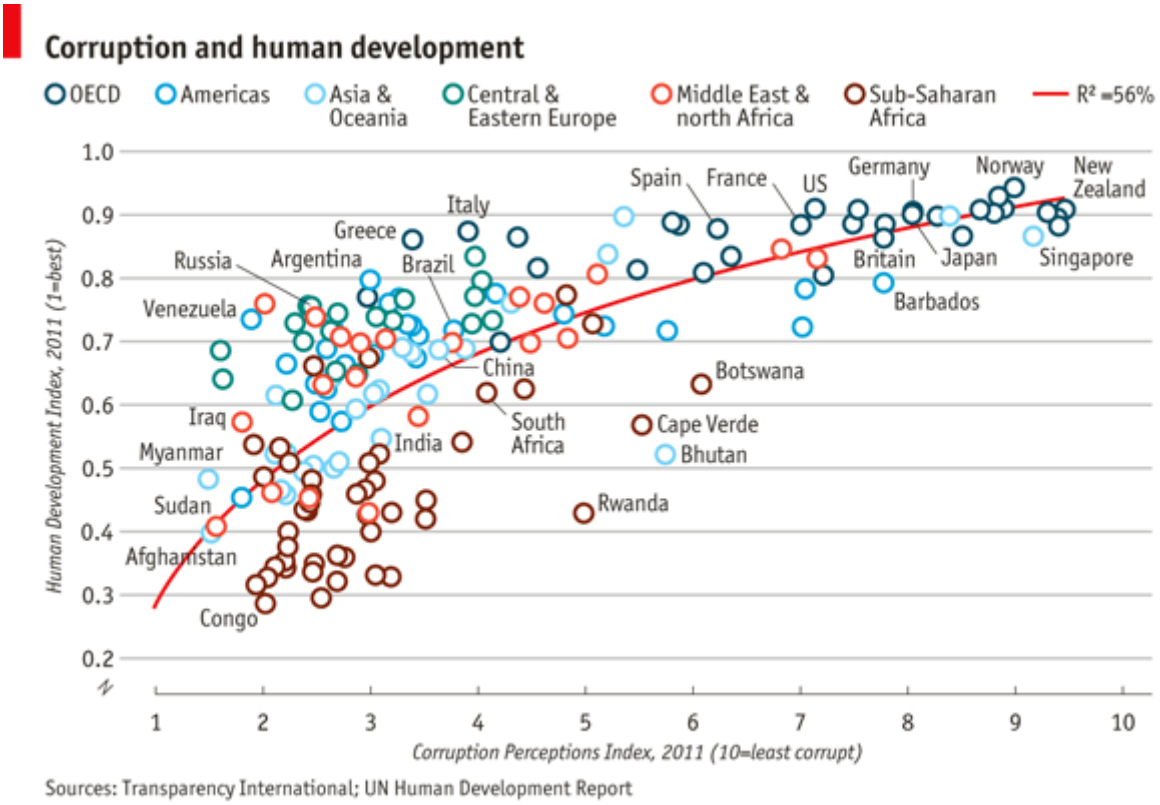
Corruption and human development
OECD
Americas
Asia & Oceania
Central & Eastern Europe
Middle East & north Africa
Sub-Saharan Africa
R² =56%
Human Development Index, 2011 (1=best)
Corruption Perceptions Index, 2011 (10=least corrupt)
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
1
2
3
4
5
6
7
8
9
10
Spain
France
US
Germany
Norway
New Zealand
Italy
Greece
Russia
Argentina
Brazil
Britain
Japan
Singapore
Venezuela
Barbados
China
Botswana
Iraq
South Africa
Cape Verde
India
Bhutan
Myanmar
Sudan
Rwanda
Afghanistan
Congo
Sources: Transparency International; UN Human Development Report

Figure 3:

What Is The Grammar Of Graphics?

The basic idea: independently specify plot building blocks and combine them to create just about any kind of graphical display you want. Building blocks of a graph include:

- data
- aesthetic mapping
- geometric object
- statistical transformations
- scales
- coordinate system
- position adjustments
- faceting

Example Data: Housing prices

Let's look at housing prices.

```
housing <- read.csv("data/landdata-states.csv")
head(housing[1:5])
```

```
##   State region Date Home.Value Structure.Cost
## 1    AK   West 2010     224952         160599
## 2    AK   West 2010     225511         160252
## 3    AK   West 2010     225820         163791
## 4    AK   West 2010     224994         161787
## 5    AK   West 2008     234590         155400
## 6    AK   West 2008     233714         157458
```