R programming

José R. Valverde
jrvalverde@cnb.csic.es
CNB/CSIC

Command line practical session

- Did you hear the joke about statisticians?
- Probably...

Contents

- Data analysis
- Read and write data
- Data manipulation
- Graphics

Get pre-existing data

- `data(iris)`
- `help(iris)`
- R comes with some datasets already installed. Moreover, most packages (collections of functions provided by others) also come with their own additional datasets
- We can load these datasets using function “data( )' by providing the name of the dataset.
- If the dataset belongs to a specific “package', we'll need to indicate the “package' name as well. More on this later.

Basic descriptives

- **`iris`**
  - This will show the dataset; if it is too large we may not be able to see it entirely
- **`summary(iris)`**
  - This is more useful: it gives us summary information on all the variables of the dataset
- **`iris$Sepal.Width`**
- **`iris$Petal.Length`**
  - This will show the values in the correspondingly named columns of the dataset
- **`table( iris$Species )`**

Hypothesis testing

```
shapiro.test(iris$Sepal.Length)
bartlett.test(iris$Sepal.Length ~
                iris$Species) # ~ means'by'
t.test(iris$Sepal.Length[1:50],
       iris$Sepal.Length[51:100])
wilcox.test(iris$Sepal.Length[1:50],
       iris$Sepal.Length[51:100])
a <- aov(iris$Sepal.Length ~
         iris$Species)
TukeyHSD(a)
```

Question

- Can you tell what each of the previous functions was intended to do?

Basic graphics

```
hist( iris$Sepal.Length )
hist( iris$Sepal.Width )
hist( iris$Petal.Length )
hist( iris$Petal.Width )
boxplot( iris$Sepal.Length )
```

- Repeat all graphics with boxplot(...)

```
plot( iris$Sepal.Length,
      iris$Petal.Length )
```

- Repeat for all combinations
- Can you identify any relationships?

Modifying data, the “visual” way

```
edit(iris)
```

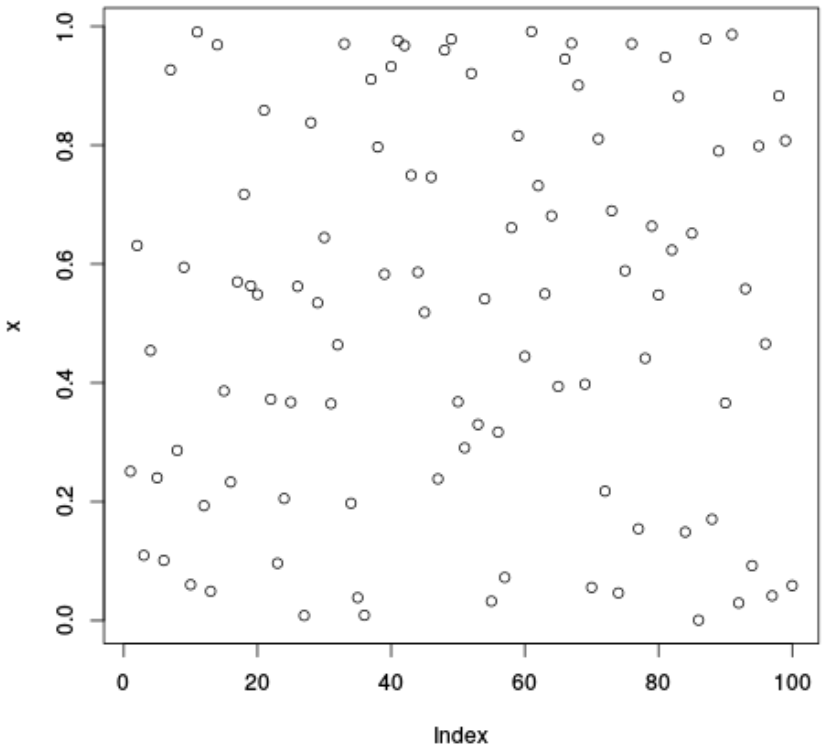
- What this does:
  - It will open a new window, where the contents of the selected variable (iris in this case) can be modified.
  - The 'edit' function will create new contents, with the modified values, and return them.
  - We can assign the values produced by *edit(iris)*, our modification of the “iris” dataset, to a new variable:

```
new_iris <- edit(iris)
```

Uniformly distributed random data

- We can generate a uniformly distributed set of values using a pseudo-random number generator such as “runif()”

```
x <- runif(100)
plot(x)
```



Since they are random, your results can be different.

Random numbers

- Random numbers are normally generated with a pseudo-random number generation (PRNG) function.
- This function normally generates new values from the previous ones.
  - **If we start from the same original value we get the same numbers**
  - If we want to obtain different numbers, we need to start from different "**seeds**"

Random numbers (2)

- Using a specific seed allows us to always obtain the same values (good for repeatability).

```
y <- runif(100)
plot(x, y)
set.seed(2023)
x <- runif(100)
set.seed(2023)
y <- runif(100)
plot(x, y)
```

- You should use a seed if you want your results to be repeatable.
  - Be conscientiously careful when using seeds.
  - Good for repatable science: use seeds and report them.

Generating normal data

- We can generate a set of random data following a specific distribution:

```
help(rnorm)
set.seed(1234)
x <- rnorm(100, 0, 1)
hist(x)
boxplot(x)
y <- 5 + 2 * x + rnorm(100, 0, 1)
plot(x, y)
```

- *Notice we make lots of plots. It is a good idea to make sure your supposedly random data does indeed look random (or at least, shows no obvious non-uniformities)*

Generating normal data explained

- We started by setting a “seed' for the *pseudo*-random number generator
- We then generate 100 values picked at random from a normal distribution with mean 0 and standard deviation 1
  - rnorm(100, mean=0, sd=1)
- And do some plots to see if it looks convincing (*notice that when few data is generated it may strongly deviate from the expected distribution and still be valid).*
- Next we generate a new set of values y from x, where y = 5 + 2 · x + some random noise
- And plot them

Customizing graphics

- You can add elements to a graphic, for instance, abline(x, y) adds a line with intercept x and slope y.
- In the previous example we computed y as a line y = 5 + 2 · x + some random gaussian noise. The command:

```
abline(5,2)
```

- Will add the corresponding reference line

Additional graphic customization

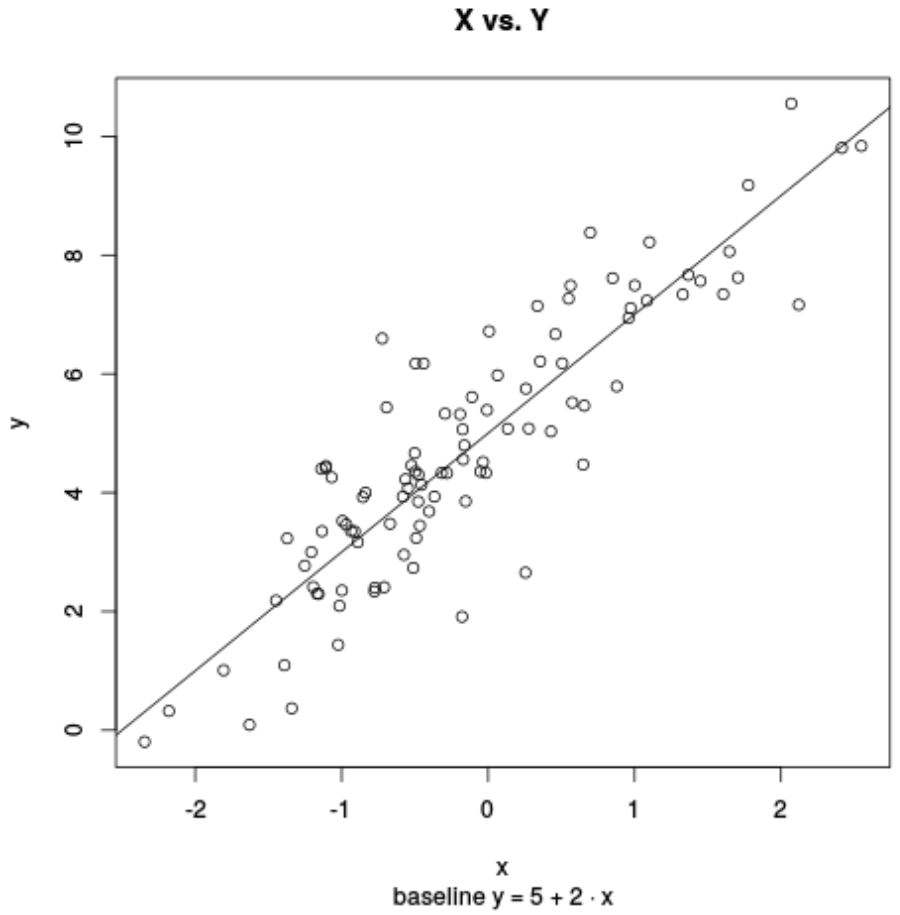
- **points**(x, y)
  - Add points at coordinates x, y
- **text**(x, y, text)
  - Add some text placed at the x,y coordinates
- **lines**(x, y)
  - Add lines connecting the points given in the vectors x and y
- **segments**(x1, y1, x2, y2)
  - Add segments from (x1, y1) to (x2, y2)

Take note of these commands and the ones in the next slide, for you will want to use them frequently.

Additional graphic customization (2)

- **arrows**(x1, y1, x2, y2)
  - Add arrows from (x1, y1) to (x2, y2)
- **abline**(a, b)
  - Add a baseline with intercept a and slope b
- **legend**(x, y, text)
  - Add a legend at the specified (x, y) position
- **title**(“title', “subtitle')
  - Add a title and a subtitle

Exercise



Try to generate this plot

Did it look like this?

```
x <- rnorm(100, 0, 1)
 y <- 5 + 2 * x + rnorm(100, 0, 1)
plot(x, y)
abline(5,2)
title("X vs. Y", "baseline y = 5 + 2 * x")
```

Now try to repeat it

- If you didn't use a seed, you can't.

Copying graphics to a file

- We can save any graphics that we have generated.
- R provides various graphics “devices'. By default it will draw to our screen.
- Some "devices" can be attached to files
- We can copy a graphic to a file by sending it to the correct "device". Here we copy the current graphic to a file named 'plot.png':

```
dev.copy(png, file='plot.png')
dev.off()
```

- If you can't find it, `getwd()` will show youin which directory/folder you are working now.

Copying graphics explained

- **dev.copy(png, file='plot.png')**
- **dev.off()**
- That means: make a copy of the current graphic (the one we see on the screen) into a new device (dev.copy)
- The device chosen is a device that makes a PNG graphic, and will save it on file “*plot.png*'
- After writing the file, we "turn it off". If we do not, the PNG “device' will remain open, and subsequent plots will overwrite the previous one (both in screen and in the file)!
- We cannot see the file contents until the PNG device is turned off (they are not saved until it is turned off, since there may be additional plots)

Copying graphics (2)

```
hist(x)
dev.copy(png, "plot.png")
```

- Try to see the contents of “plot.png”

```
boxplot(x)
```

- Try to see the contents of “plot.png”

```
plot(x, y)
dev.off()
```

- Can you tell what does “plot.png” contain before looking at it yourself?
- The histogram, the boxplot or the x-y scatterplot?

Saving graphics

```
# This will not show the graphic on
# the screen
png("plot.png")
hist(x)
```

We can specify the output device: the png() command tells R to use a PNG file instead of the screen.

```
dev.off()
```

- This will close the last open device (png) and now graphics will go back to the screen again.

Graphic parameters

- Did you find these plots boring?
- You can further customize the appearance of the graphics display by specifying a number of parameters.
- Display parameters allow you to control how things are drawn on the selected graphics device (by default, your screen).
- You define these parameters with par( )

```
help(par)
```

Changing graphics parameters

- We change the display properties using function par( ).
- Since each device may use different parameters, we need to modify them explicitly by name

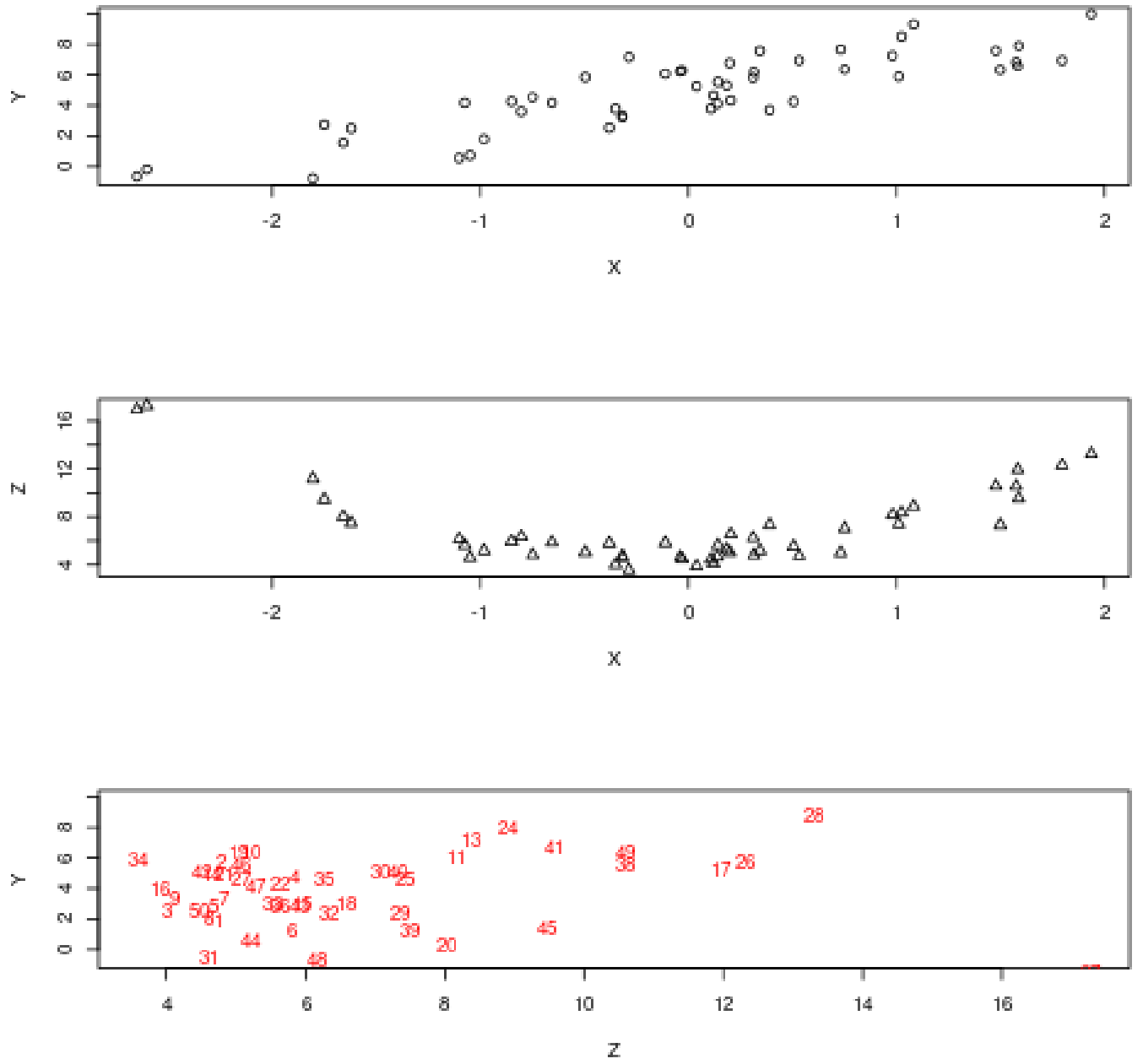
```
par(mfrow = c(1, 2))
```

  - mfrow allows us to plot several graphics on the same device (a screen window or a file...).
  - This example splits the device in one row and two columns (remember, matricial data is arranged by rows first)

```
hist(x)
boxplot(x)
```

3 in 1

```
x <- rnorm(50, 0, 1)
y <- 5 + 2 * x + rnorm(50, 0, 1.3)
z <- 5 + x + 2 * x ^ 2 + rnorm(50,0,1)
par( mfrow = c(3, 1))
plot(x, y, xlab='X', ylab='Y')
plot(x, z, xlab='X', ylab='Z', pch=2)
plot(z, y, xlab='Z', ylab='Y', type='n')
text(z, y, labels= as.character(1:50),
     col='red',
     pos=rep(1, 50), offset=0.6)
```

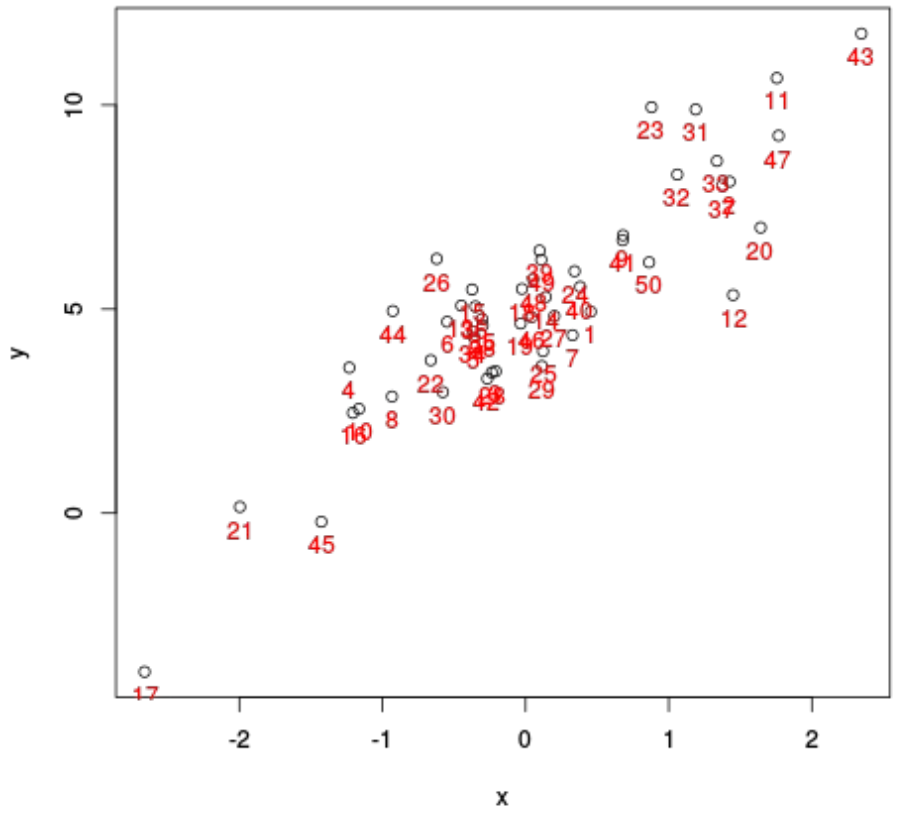

Y
X
Z
X
Y
Z

3 in 1 explained

- We define a plot area composed of 3 rows and 1 column
- The first plot shows x vs. y
- The second plot shows y vs. z using a different plotting character (pch)
- The third plot shows z vs y but draws no values (type=”n”)
- Then we add to the last plot (the third) text labels for the z,y values using as label the corresponding number
- “col” specifies the colour, “pos” is the position, and “offset” the displacement in the direction indicated by “pos”

Exercise

- Set the display layout back to one single plot
- Plot x vs. y as dots
- Add labels, using different values for “pos” (from 1 to 4) to see which direction corresponds to each “pos” value.
- Redraw the plot each time to see clearly the four directions
- Repeat with a different value for “offset”


y
x
10
5
0
-2
-1
0
1
2

Explore associations

- Let's go back to the “iris” data
- Visualize scatterplots of each of the continuous variables against each other
  - Sepal.Length
  - Petal.Length
  - Sepal.Width
  - Petal.Width

```
plot(iris$Sepal.Length,
     iris$Sepal.Width)
```

Exercise

- Try to draw a **trellis graphic** of the plots.
- *Trellis graphics* are simply a panel showing all comparisons tabulated in two dimensions:
- We have four groups, two sepal and two petal.
- Let's plot the Sepal groups against each other.
- Define a drawing panel of 2 x 2 graphics
- Plot all 4 comparisons in order

SL/SL SL/SW
SW/SL SW/SW
where
SL:Sepal.Length
SW:Sepal.Width

Fixing display limits

- Everytime you draw a plot, it may happen that its optimal size (as computed by R) exceeds the size of the window or device you want to draw it in. You'll get:
  - *Error in plot.new(): figure margins too large*
- To fix it, find out the values of parameter "mar"
  - `par("mar")`
- Then set all values to 1. Let us assume it has 16 values, then we should use
  - `par(mar=rep(1, 16))`

Visualize factor effects

- Looking at the SL/SW plots, it seems as if there are various parallel linear correlations. This may happen when we have a mixture of subgroups (e.g. age vs height in a mixture of men and women).
- Species is a factorial variable. We can try to see the association of each continuous variable to this factor:

```
par(mfrow=c(1,1))
car::Boxplot(iris$Sepal.Length ~
                iris$Species)
```

- This plots Sepal.Length using separate plots for each value of iris$Species (the ~ means "by")
- Notice the function name: we need to use *car::Boxplot( )*, not “boxplot()” this time. More on this later

Feeling sleepy?

- Load the “sleep” dataset

  data(sleep)

  help(sleep)

  sleep

  summary(sleep)

  car::Boxplot(sleep$extra ~ sleep$group)

  attach(sleep)

  car::Boxplot(extra ~ group)

- Do you think that the difference between groups is significant?

Statistics!

```
t.test(extra ~ group,
       alternative="two.sided",
       conf.level=.95,
       var.equal=TRUE)
```

- Means: test the model “extra varies depending on the treatment group”
- The *null hypothesis* is that there is no difference.
- The *alternative hypothesis* is that there is a difference between groups (be it positive or negative, i.e. *two-sided*)
- The confidence level is set to 0.95
- Variances are assumed to be equal

- Can you confirm a difference?

More t-tests

- Try to use different alternative hypotheses:
  - “greater”
  - “less”
- Try assuming non-equal variances (var.equal=FALSE)
- Can you confirm a difference now?
- Can you believe it?

Data manipulation

- The problem here – **`help(sleep)`** – is that we took measures on the same 10 patients using two different drugs.
- Data is, therefore, paired.
- R is assuming we are working with two different and independent experiments because we gave the values all stacked in one column – **sleep** –
- We need to rearrange the data into two paired columns so R knows they are paired observations and uses the paired-samples t-test

Un"stack"ing data

```
pairsleep <- unstack(sleep,
                     extra ~ group)
pairsleep
attach(pairsleep)
t.test(X1, X2,
            alternate="two.sided",
           conf.level=0.95,
            paired=TRUE)
```

- How does it look now?

Testing assumptions

- In parametric tests we often assume normality and homocedasticity. We should always check our assumptions:
- Quantile-quantile plot for normality:

```
qqline(iris$Petal.Length)
with(iris, qqnorm(Petal.Length,
                  ylab="Petal length"))
```

- Shapiro-Wilk test for normality:

```
shapiro.test(iris$Petal.Length)
```

- Variance homogeneity

```
bartlett.test(Petal.Length ~ Species, data=iris)
```

- Note that we have used various “simplifying tricks” here. Try them and see if you understand what they are doing.

Exercice

- Generate values from a random normal distribution and test if the random values selected are representatively normal
  - Try with 10, 50, 100, 500 and 1000 values
  - Check using a Q-Q plot and the Shapiro-Wilk test
- Generate two sets of values with the same variance (specified through the standard deviation as parameter “sd=” to rnorm( )) and a third set with a different variance
  - Test for homogeneity of variances using Barlett's test.

Coming up next

- Writing scripts
- No more boring repetitions
- Automating your work...

Thank you

Questions?