R programming

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

Scripting

Statistical Facts

Flying on a plane is indeed secure: almost all deceased in plane accidents died after reaching the floor.

Scripting

Why

- We have already used a lot of commands
- Do you remember each one and all of them?
  - If you don't, then welcome to the select group of advanced programmers.
  - If you do, don't worry, you will forget very soon and then you will be welcome in the select group of advanced programmers.
- Advanced programmers do not remember everything, nor do they like to write commands. *That is what computers are for.*

The "recommended" organization

- For a given project, e.g, "my_project":
  - Folder "**my_project/**" contains
    - folder "**R/**" with the commands used to do the analyses stored as scripts "./my_project/R"
      - **script-001.R** **"./**my_project/R/script_001.R"
      - **script-002.R** **"./**my_project/R/script_002.R"
      - ...
    - folder "**data/**" with the corresponding data files used in the analyses
      - **dataset-001.tsv** **"./**my_project/data/dataset_001.tsv"
      - **dataset-002.Rdat** **"./**my_project/data/dataset_002.Rdat"
      - **dataset-003.csv** **"./**my_project/data/dataset_003.Csv"
      - ...
  - This requires longer filenames, but is cleaner.

Putting commands in a file

- If you do not want to repeat all the commands each time, then there is a simple solution
- Collect all the commands in a file
  - This must be a **text** file (not a word processor file)
- Whenever you need to, ask R to repeat the work in that file
  - Instead of typing the commands all over again
- For instance:
  - copy and paste the text in the following slide in a text editor and save it as "script-001.R".

script-001.R

```
data(iris)
dim(iris)
names(iris)
row.names(iris)
with(iris, qqnorm(Petal.Length,
     ylab="Petal length"))
qqline(iris$Petal.Length)
shapiro.test(iris$Petal.Length)
bartlett.test(Petal.Length ~ Species,
              data=iris)
```

Running a script

- It is easy: use the 'source( )', Luke...
- If you saved the former commands in a text file and named the file, for example, 'script.R', then it suffices to type within R

```
source('script-001.R')
```

- Or you can directly use it with Rscript

```
Rscript script-001.R
```

- And, if your computer is properly configured, you may even just double-click on the file.

Reinforcing feeble memories

- If you still remember what this script did, you are lucky
- Even so, you will not remember one week, one month or one year from now.
- For this reason it is highly recommended that you add notes to your commands explaining what they do
- These notes are preceded by a # and are called '**comments**'

Comments

- R will ignore your commens (i.e. whenever R finds a # in a line, it will ignore *the rest of the line*)
- This allows you to *document* your **code** (the commands you write for R)
- You are **strongly encouraged** to extensively document everything you do.
- Best done while you write it, before you forget...
  - Copy and paste the contents of the next slide in a text editor and save it as "**script-002.R**".

```
# R comes with a number of standard datasets for us to try
# we will make use of one of them in this script
# iris gives the measurements in centimeters of the variables
#      sepal length and width and petal length and width,
#      for 50 flowers from each of 3 species of iris.
data(iris)

dim(iris)   # get the dimensions of the data set
names(iris)    # get the column (variable) names
row.names(iris) # get the row names

# test for normality
shapiro.test(iris$Petal.Length)
# test for homogeneity of variances
bartlett.test(Petal.Length ~ Species, data=iris)
# draw a Q-Q normality plot
with(iris, qqnorm(Petal.Length, ylab="Petal length"))
qqline(iris$Petal.Length)
```

Practical advice

- As a rule, it is better to organize your analyses in directories/folders
- A common convention is to create a folder for each **project** (a set of related analyses)
- Then
  - inside the project folder, create another folder called "**R**" where you will store all the scripts used to run the analyses
  - inside the project folder, create a second folder (at the same level as the "R" folder) named "**data**", where you will store all the data used.

Getting data into R

Preparation

- Download the archive with the R exercises data files
- **exercises/R_exercises_datafiles.zip**
- Create a directory for it
- Extract the contents in said directory
- Make the directory your working directory
  - Launch R
- Or launch R and make that directory your working directory
  - using **`setwd('..how to go there..')`**

Statistical data

- The easiest, and most common, way of formatting data is by laying it out as a table of rows by columns.
  - This is what spreadsheets and databases do
- Typically, we will want to assign names to rows and columns for understandability

plot	place	treatment	height	yield
1	Toledo	Control	100	10
2	Toledo	Pesticide	90	8
3	Cadiz	Control	110	12
4	Cadiz	Pesticide	85	9

Data exchange

- The traditional and most pervasive method is using “delimited data”:
  - each row uses one line
  - may or not include a header with column names
  - values in a row are separated by a “delimiter” (comma, semicolon, tabulator, space...)
  - when the delimiter may appear in values, values may be “quoted” (single or double)
  - numbers, alas, are not uniform, but specific, e.g.:
    - US; decimals are separated by dots
    - ES: decimals are separated by commas

Getting data from a file

- You will need to specify
  - The *separation* character between fields (TAB, comma, white space, etc...'
  - The *enclosing* character for fields (single or double quotes, etc...), if any.
  - Any *localization* information, such as the character used to separate the integer and decimal parts in numbers.
  - Whether the file includes a *header* with column names; and possibly row names in one of the columns.
  - Other needed details (e.g. special values).

Examples

- Header: yes; field delimiter: none; separator: TAB; localization: English (decimal is '.')

```
id   group  time1  time2  time3  time4
1    A      31.0   29.0   15.0   26.0
2    A      24.0   28.0   20.0   32.0
```

- Header: yes; field delimiter: "; separator: ','; localization: Spanish (decimal is ',')

```
"id","group","time1","time2","time3","time4"
"1","A","31,0","29,0","15,0","26,0"
"2","A","24,0","28,0","20,0","32,0"
```

A sample file

id	group	time1	time2	time3	time4
1	A	31	29	15	26
2	A	24	28	20	32
3	A	14	20	28	30
4	B	38	34	30	34
5	B	25	29	25	N/A
6	B	30	28	16	34

- Copy and paste this and save it to a file 'twisk.txt' using "Save as..." and selecting "Text only".

Reading a table of data

```
twisk <- read.table('twisk.txt',
                    header=TRUE,
                    row.names=1)
twisk
```

- This reads a TAB-separated data set from the specified file, using the first line as a header that contains all the column names, and the first column as a list of row names
- By default, it assumes that fields are separated by white space (a tabulator is considered white space).

Hint

- You can use a URL instead of a file name.
  - table <- read.table("http://example.net/file.tsv")
- file.choose() is a function that allows you to choose a file and produces its name. You can use it instead of the file name:

```
twisk <- read.table(file.choose(),
                    header=TRUE,
                    row.names=1)
```

Dealing with missing data

- Often, when collecting data, we find situations were an observation cannot be obtained. This observation must be distinguished as 'Not Available'.
- Different experimenters may decide to encode non-available data in different ways (e.g. as an impossible or negative value or as a specific string like 'N/A')
- We can identify these values with '*na.strings*='

```
twisk <- read.table('twisk.txt',
                    header=TRUE,
                    row.names=1,
                    na.strings='N/A')
```

005-read.table_NA.R

Using CSV data

- Most spreadsheets will export data for use in other programs as 'CSV' (comma-separated-value) files

```
Id,group,time1,time2,time3,time4
1,A,31,29,15,26
2,A,24,28,20,32
3,A,14,20,28,30
4,B,38,34,30,34
5,B,25,29,25,N/A
6,B,30,28,16,34
```

- Copy and paste this and save it to a text file called "**twisk.csv**"

```
twisk <- read.csv('twisk.csv', header=TRUE,
                  na.strings='N/A')
```

006-read.csv.R

Hint

- You can also use URLs in most file access functions: copy and paste this

```
twisk <- read.csv(
'https://raw.githubusercontent.com/
jrvalverde/Rprog/main/data/
twisk.csv',
    header=TRUE,
    na.strings='N/A')
```

Opening an Excel file

```
library(xlsx)

cherry <- read.xlsx(file.choose(), sheetIndex=1)
```

- This will open the chosen Excel (.xlsx) file and load the first spreadsheet in the file as “cherry”
- Inspect the contents

```
head(cherry)

cherry

plot(cherry)

summary(cherry)
```

NOTE: if you do not have an R extension you can add it with install.packages("name"), for instance: install.packages("xlsx") or better install.packages("xlsx", dependencies=TRUE)

008-read.xlsx.R

)

Hint: extending R

- There are many extensions to R
- To install any of them:
  - Use google to find out which one you want
  - if it is in CRAN

    install.packages(‘name’, dependencies=T)
  - otherwise, follow the instructions, e.g.

    BiocManager::install(‘name’)

    install_github(‘repo’, ‘user)

    etc...

Defining factor variables

- **read.table()** will treat any column with literal (non-numeric) values as composed of 'factors' or categorical variables.
- If you encoded a factor using numbers (e.g. 1 / 2 for sex M / F) you will need to tell R thhey are not numbers using *factor( )*
- *factor( )* converts its argument into a factor

```
v <- factor(c(1,1,1,1,2,2,2,2))
print(v)   # note the difference!
```

Selecting subsets

- `cherry <- read.xlsx(file.choose(), sheetIndex=1)`
- `cherry[3,]` *# third row*
- `cherry[3:5,]` *# rows 3 to 5*
- `cherry[-c(2,4),]` *# all rows except # rows 2 and 4*
- Now try the following and see if you can understand what you get after each call.
- We are not assigning the result to a variable, but you certainly can, if you want to use the subset afterwards

009-subindex_selection.R

Subsets

```
subset(cherry, Height>70)
subset(cherry, Height>=70)
subset(cherry, Height==80)
subset(cherry, Height==80,
       select=c(Girth,Volume))
subset(cherry, Height>80 & Girth>15)
subset(cherry, Height>80 | Girth>15)
```

010-subset.R

Merging data

- Sometimes data comes from different sources and we need to merge it into a new dataset to facilitate analysis. Try the following and see what happens

```
newData <- data.frame(Girth=c(11.5, 17.0),
                      Height=c(71, 75),
                      Volume=c(22, 40))
newData
allData <- rbind(cherry, newData)
allData
```

- What does each command do?

011-rbind.R

Merging data from various sources

- Let us assume we have data from other source (we'll simply make it up this time)

```
precipitation <- rnorm(n=31, mean=50, sd=10)

precipitation

allData2 <- cbind(cherry, precipitation)

allData2
```

- Obviously, variables should have the same order in both data sets.

Example
A (small) real dataset

Statistical facts
Natality rate is double the mortality rate. Therefore one out of every two persons is immortal.

A proteomic dataset

- We have a control and three variants.
  - 10 control samples, 2 of variant 1 and 1 of variants 2 and 3
- Are there differences between the variants?
- If so, which are the differences?
- We'll use the data in file "*proteomics.tab*"

```
data <- read.table("proteomics.tab",
                   header=T,
                   row.names=1)
```

day_02-analysis.R

A proteomic dataset

- We have a control and three variants.
  - 10 control samples, 2 of variant 1 and 1 of variants 2 and 3
- Are there differences between the variants?
- If so, which are the differences?
- We'll use the data in file "*proteomics.tab*"

```
data <-
read.table("https://github.com/jrvalverde
/Rprog/raw/main/data/proteomics.tab",
                        header=T,
                        row.names=1)
```

day_02-analysis.R

The dataset

```
head(data) ; str(data)
```

- We have data from
  - 10 control experiments (control.1 to control.10)
  - 2 experiments on variant 1 (A and B)
  - 1 experiment from variant.2 and another from variant.3
- For each strain, we have measured 312 different parameters (proteins).

Transforming data

- We currently have data organized in columns by sample
- But we want each sample in a row: we need to transpose the dataset
- `tdata <- t(data)`
- Now, each column corresponds to one measure and we can use (e.g. for measure 1)
- `shapiro.test(tdata[,1])`

Parametric or not

- Our first decision is which kind of tests shall we apply.
- A first hypothesis may be that protein observations follow a normal distribution
- We need to check each protein/measure and verify that its observations in the control group do indeed follow a normal distribution.

Flow control

Logical operations

==	equal
!=	not equal
> <	greater/less than
>= <=	greater/less than or equal
&	and
|	or
!	not

- *Write this down somewhere to remember later.*

if

Conditional execution

- **if** (cond == true) cmd
- **if** (cond == true) cmd1 **else** cmd2
- Example

```
if (1 == 0) print(1) else print(2)
```

- "If" operates on length-one logical vectors (i.e. tests a **single** value)

Statements and compound statements

- A single command is called a **statement**
- You can group several commands together as if they were a single one with { and }
- This new, multi-command, group is called a **compound statement**
- Thus
  - **if** (condition == true) stmt1 **else** stmt2

  is conceptually the same as
  - **if** (condition == true) { stmt1, stmt2…} **else** { stmt3, stmt4... }

Vectorial conditions

- **ifelse**(test, true_value, false_value)
- This function operates on vectors

```
x <- 1:10     # Create sample data
ifelse(x<3 | x>8, x, 0)
```

- This takes a vector (x) and tests each element of the vector to see if it is less than 3 or more than 8. If it is, the value is left untouched, otherwise, it is substituted by a zero.
  - We are actually eliminating central values.
- We have already seen other short-hand versions of this method
  - If you don't remember them it may be time to revisit previous presentations and save the recipes in a script (with explanatory comments) for future reference.

for

Iteration over a list of values

- **for** (variable **in** sequence) statement
- Example

```
mydf <- iris
# seq(along=...) 1:n along the values in...
for(i in seq(along=mydf$species)) {
    cat("value", i, "is", mydf[i,1], '\n')
}
```

  - In practice, it is better if you always use { } to enclose the statements in conditionals and loops, even if you only have one statement.
    - It is more visible and easier if you decide to add more commands later

015-for.R

Zeroing values

- We can also use **ifelse**, but let us see it with a **for** loop.
- In the next example we will try to remove values greater than 5 from a vector
  - First we create a vector with 10 sequential values
  - Then, for all the values in the vector we will
    - test if it is less than or equal to 5
      - if it is, we add it to the end of another vector, z
      - if it is not, we put a zero instead at the end of z

Zeroing values (1)

```
x <- 1:10
seq(along=x)    # verify first that we do it right
z <- c()        # start with 'z' empty: c()
for(i in seq(along=x)) {
    if (x[i] <= 5) {
        z[i] <- x[i]  # place x[i] after z
    } else {
        z[i] <-  0     # place zero after z
    }
}
print(z)
```

Hint

- When you want to know what is going on, use the **print()** command to print values and see how they change:

```
x <- 1:10
z <- c()          # create an empty vector
for(i in seq(along=x)) {
    print(i)
    if (x[i] <= 5) {
        z[i] <- x[i]
    } else {
        z[i] <- 0
    }
    print(z)
}
```

017-for-if-else.R

Zeroing values (2)

- Using **ifelse**, it would become

```
# from 1 to 7 and vice versa
x <- c( 1:7, 7:1 )
# if x <= 5, the result is x,
# otherwise, it is 0
z <- ifelse(x <= 5, x, 0)
```

Removing values

- If we do not want the values, we could simply not add them:

```
x <- c( 1:7, 7:1)
z <- c()
for(i in seq(along=x)) {
    if (x[i] <= 5) {
        z <- c(z, x[i])
    }
}
```

- Thus, z will only contain the values which are <= 5

019-for-remove.R

Stop if values are ≤ 5

```
z <- c()     # create an empty vector
for(i in seq(along=x)) {
     if (x[i] <= 5) {
          z <- c(z, x[i])
     } else {
          stop("values need to be <= 5")
     }
}
print(z)
```

- This will *stop()* all the work and produce an ***error*** *message*, so we know that there was a problem

Warning if values ≤ 5

```
x <- 1:10
z <- c()    # create an empty vector
for(i in seq(along=x)) {
    if (x[i] <= 5) {
        z <- c(z, x[i])
    } else {
        cat("warning: 'x' values need to be <= 5\n")
        break
    }
}
print(z)
```

- Stopping everything should be a last resource. "***break***" will stop only the for loop (here we write a warning message before calling break to explain why we are cutting it short prematurely)

21-break.R

Sample real dataset (continued)

Exercise

- In the proteomics dataset:
- Do all 312 tests for normality for each variable using the shapiro.test( ) function.
  - hints:
    - use the transposed dataset (**tdata**)
    - use variables instead of "magic" numbers so your intent is clear
    - print( ) will print a summary of the test
    - cat( ) allows you to con**cat**enane text, numbers, variables... separated by commas, e.g.
      - cat("hi, I am", 25, "years old\n")
    - '\n' means "jump to a new line"

Doing it all at once

```
n.measures <- 312
for (i in 1:n.measures)
    print( shapiro.test(tdata[ ,i]) )
```

- Or, better

```
for (i in 1:n.measures) {
    cat("Analyzing normality of measure", i, '\n')
    print(shapiro.test(tdata[ ,i]))
}
```

- Or if we are in a hurry and do not care for how it looks and do not want to use the transpose

```
by.row <- 1
apply(data, by.row, shapiro.test)
```

day_02-analysis.R

Cleaning clutter

- shapiro.test() produces a **list** of values, among them the p.value, with a for loop we can do

```
for (i in 1:n.measures) {
    cat("Analyzing normality of variable",
        i, '\n')
    st <- shapiro.test(tdata[ ,i])
    cat('   p.value =', st$p.value, '\n')
}
```

day_02-analysis.R

Ultraclean

- Even better, we can combine a for loop and a condition to only print significant values (the null hypothesis is that our data is normal):

```
for (i in 1:n.measures) {
    st <- shapiro.test(tdata[ ,i])
    if (st$p.value >= 0.05)
        cat('measure', i, 'is normal\n')
}
```

day_02-analysis.R

Compare measures

- Now, repeat the analysis, but this time
  - using Wilcoxon test: wilcox.test( ) to compare control (rows 1 to 10) and first variant (rows 11 to 12, also named "variant.1.A" and "variant.1.B")
  - using a significance level < 0.1
  - printing only significant measures

A "difficult" to read solution

```
for (i in 1:312) {
    wt <- wilcox.test(tdata[1:10 , i],
                      tdata[11:12, i)
    if (wt$p.value < 0.1)
        cat('measure', i, 'has a p.value',
            wt$p.value, '\n')
}
```

day_02-analysis.R

A better solution

```
control <- 1:10     # rows 1..10
var.1 <- c("variant.1.A", "variant.1.B")
for (i in 1:n.measures) {
    wt <- wilcox.test(tdata[control ,i],
                      tdata[var.1, i)
    if (wt$p.value < 0.1)
        cat('measure', i, 'has a p.value',
            wt$p.value, '\n')
}
```

day_02-analysis.R

while

Iterate while a condition holds

- **while** (condition) statement
- Example

```
z <- 0
while(z < 5) {
    z <- z + 2
    print(z)
}
```

- Observe the output and think about it.
- Can you explain what is going on?

Hint: trace execution manually

- We start with a value of 0
- Then we check the value of z to see if it is < 5
  - The first time it is 0
  - So we add 2 to z (0+2=2) and assign it to z
  - z now is 2
  - The second time we add another 2, z becomes 4
  - The third time we add 2 more, z becomes 6
  - Now z is greater than 5 and the loop stops.
- We could pre-calculate how many times it was needed, and use a for loop, but using while allows us to repeat some work indefinitely while a condition holds.

Sample real dataset (continued)

Dealing with the most significant measures

- We need to save the p.values of all comparisons
- Then we sort the dataset according to p.value.
- Then we will print results until they are no longer significant

Saving the p.values

- We will simply save them in a new vector:

```
control <- 1:10     # rows 1..10
var.1 <- c("variant.1.A", "variant.1.B")
pvals <- c()
for (i in 1:n.measures) {
    wt <- wilcox.test(tdata[control ,i],
                      tdata[var.1, i])
    pvals <- c(pvals, wt$p.value)
}
print(pvals)
```

day_02-analysis.R

Sorting data

- Sort a vector (ascending)

```
sort(pvals)
```

- Get the indexes that would produce a sorted list

```
order(pvals)
in_order <- order(pvals)
pvals[in_order]
```

Why *order()*?

- *sort()* is a fine and very useful function. But sometimes it is not enough for our needs.
- But if we sort all columns in a data set separately, each will come out in a different order... and values will not match any more.
- We can get the *order()* of one of the columns (the *master* column) and then apply this same order to all the other columns
  - other columns will not be "sorted" (only the "ordered" **ones**) but values will match across.

Sorting matrices and data frames

- To sort a matrix or data frame according to some columns, we can use order( )
- This allows for complex sorting: to sort according to sex and age we could use
  - index <- order(data$sex, data$age)
- and then apply that order to the data
  - sorted_data <- data[order,]
- For instance

```
idx <- order(pvals)
# now sort all the columns (measures) using it
p.sorted.tdata <- tdata[ , idx]
# or rows in data
p.sorted.data <- data[idx, ]
```

day_02-analysis.R

Print significant measures

- We now want to print measures with $p < 0.1$ in order, but we do not know how many there are.
- We need a while loop and a condition
- Can you try?

A possible solution

```
i <- 1
while p.val[i] > 0.1 {
    cat("P.value of measure", i, "is",
        p.val[i], '\n')
    i <- i + 1
}
```

- Remember, colnames(tdata) gives column names: try

```
i <- 1
col <- colnames(p.sorted.tdata)
while p.val[i] > 0.1 {
    cat("P.value of measure", col[i], "is",
        p.val[i], '\n')
    i <- i + 1
}
```

day_02-analysis.R

repeat

Iterate indefinitely

- **repeat** statement
- The statement is repeated forever unless break( ) is called
- Example

```
z <- 0
repeat {
    z <- z + 1
    print(z)
    if(z >= 99) break
}
```

- You better set a safe condition to stop the loop or it will never stop!

023-repeat.R

Hint

- We could have simply checked that z was 99
  - if (z == 99)
- This would have worked in this case, but is generally a **very bad** idea.
- Suppose we were using real numbers and adding a fraction
- Then it might never become exactly 99.
- **Never check for exact values in a loop!**

(27/7)

```
z <- 0
repeat {
    z <- z + (27/7)
    print(z)
    if(z >= 99) break
}
```

- That was obvious, it never reaches exactly 99, as expected, and stops in the first value greater or equal to 99 (100.2857)

(1/3)

```
z <- 0
repeat {
    z <- z + (1/3)
    print(z)
    if(z >= 99) break
}
```

- But, what about this?
- Here, one of the values is 99 and yet R does not stop and goes on to the next value (99.3333). What is going on?
- It is a problem with precision: the value calculated is almost 99 but not quite by a tiny bit. It's so small a difference that R shows it as 99, but it actually isn't. **99 never really happens here!**

apply

Working with collections of values

- We can process all the values in a collection using a loop

```
x <- rnorm(10)

for (i in 1:10) print(x[i])
```

- But R provides a more convenient way to work with collections of data values: apply( )
- **apply**(object, dim, function)
- This will apply **function** to the values in the specified **object** (a matrix or data.frame). **dim** can be 1 or 2 depending on whether we want to act on rows or columns.

Working with collections

```
apply( iris[ ,1:4], 2, mean)
```

- Here, we select columns 1 to 4 of the dataset iris, and tell R to apply the function **mean** to each column (2).

```
apply( iris[ ,1:4], 2, median)
```

```
apply( iris[ ,1:4], 2, var)
```

```
apply( iris[ ,1:4], 2, var) ^2
```

Hint: Use variables for legibilty

```
by.row <- 1 ; by.column <- 2
means <- apply( iris[ ,1:4], by.column,
                mean)
print(means)
```

- This is the same as before, but it is easier to understand. And therefore it will be more difficult to make errors.

```
medians   <- apply( iris[ ,1:4], by.column,
                    median)
variances <- apply( iris[ ,1:4], by.column,
                    var)
medians ; variances ; variances ^ 0.5 # sd
```

Processing collections by factor

- tapply(vector, factor, function)
- aggregate(x, by, function)
- Example

```
## Computes mean values of vector aggregates
#  defined by a factor (i.e. by type)
for (i in 1:4) print(tapply(as.vector(iris[,i]),
                            iris$Species, mean))

## The aggregate function provides related
# functionality
aggregate(iris[,1:4], list(iris$Species), mean)
```

029-tapply-aggregate.R

For vectors and lists

- lapply returns a **l**ist
- sapply returns the **s**implest object possible: a vector or a matrix
- Examples

```
## Creates a sample list
mylist <- as.list(iris[1:3,1:3])
mylist
## Compute sum of each list component and return result
# as list
lapply(mylist, sum)
## Compute sum of each list component and return result
# as vector (the simplest form)
sapply(mylist, sum)
```

030-lapply-sapply.R

Sample real dataset (continued)

Again, using apply

- We can repeat simplw analyses now using apply.
- We do not even need the transpose as we can work by rows:

```
by.row <- 1

by.column <- 2

apply(data, by.row, shapiro.test)
```

day_02-analysis.R

But, what about Wilcoxon's test?

- **apply( )** only takes one data argument, but wilcox.test compares two datasets. Could we do it?
- Both data sets (control and variant.1) are in the same dataset, so the data is all there.
  - If only we had a function that could run wilcox.test on columns 1:10 and 11:12 of a dataset, we could use

    apply(data, by.row, compare.control.variant)
  - and be done.

Coming next: functions!

- Try this:

```
by.row <- 1
apply(data, by.row,
      function(x) wilcox.test(x[1:10], x[1:11]))
```

Thanks

Questions?