



# ELEMENTS OF DATA SCIENCE AND STATISTICAL LEARNING

SPRING 2020

Week 2

# OUTLINE

- More R
  - Lists, dataframes, factors
  - Conditionals and loops
  - Functions
  - Loading data
- Exploratory analysis
  - Boxplots, scatterplots
- Statistical learning: what is it about?
  - Statistical modes
  - Model bias
  - Prediction vs inference
  - Prediction accuracy

# LISTS IN R

- Vectors and matrices are good for storing homogeneous data (e.g. all values are numeric)
  - Thus vectors/matrices are close to what in most other languages is called an “array” (one- or two-dimensional, respectively). The only difference is that in most languages “raw” arrays have fixed size, while in R vectors/matrices can grow dynamically (e.g. by appending values)
- R also provides a container for storing heterogeneous/arbitrary data, a **list** (thus similar to a “list” in Python, List<Object> in Java or array/vector of void \* pointers in C/C++)
  - A value of any type can be stored as an element of a list, regardless of what other elements contain: a (vector of) integers, a (matrix of) characters, another list, a function (yes, functions in R are first-class objects), etc

```
> a <- list()
> a[[1]] <- "first el"
> a[[2]] <- 2
> a[[5]] <- 1:3
```

```
> a
[[1]]
[1] "first el"
[[2]]
[1] 2
[[3]]
NULL
[[4]]
NULL
[[5]]
[1] 1 2 3
```

# LIST INDEXING

- There is another indexing operator in R, used for lists: `[[ ]]`
  - Does *NOT* allow vectors of length  $> 1$  as indexes; single index *ONLY* (technically, it works on vectors/matrices as well, but it is not of much use there)
  - The indexing operator `[ ]` preserves the mode (type) of the object being indexed. Subsetting a vector/matrix with `[ ]` returns a vector/matrix; since everything is a vector (even a “single value”), there is nothing to “unwrap”: just take a subset of vector elements, it does not matter one or many – it’s still a vector
  - Operator `[ ]` will also work on lists, but will return the object of the same mode, a *list*, even if index has length 1 (will return *list* with one element in that case). Thus: use `[ ]` on lists when you need to take a sublist.
  - Operator `[[ ]]` extracts the value (element) from the list at the specified position.

```
> x=list("first el",2,1:3); x
[[1]]
[1] "first el"

[[2]]
[1] 2

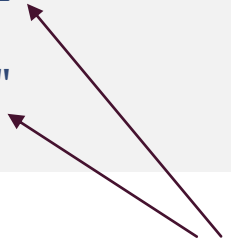
[[3]]
[1] 1 2 3
```

```
> mode(x[1])
[1] "list"
> x[1]
[[1]]
[1] "first el"

> x[1:2]
[[1]]
[1] "first el"

[[2]]
[1] 2
```

```
> mode(x[[1]])
[1] "character"
> x[[1]]
[1] "first el"
```



!!!!!!!!!!

# NAMED LIST ELEMENTS

- Similarly to vectors/matrices, elements of a list can be named
  - Special syntax can be used at the time of list creation;
  - the same function `names()` can be used on lists too!
  - Note that `names(x)` returns a *vector* of names, which can be further subsetting (even on the left hand side of an assignment as it is done in our example!); this is not unique to lists – works same way on any objects with named elements (vectors, matrices)

```
> x=list(a=1,b=c(5,-3,2),matrix(1:4,ncol=2))
> x
$a
[1] 1

$b
[1] 5 -3 2

[[3]]
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
> names(x)[3]<-"c"
> x
$a
[1] 1

$b
[1] 5 -3 2

$c
      [,1] [,2]
[1,]    1    3
[2,]    2    4
```

# LIST INDEXING BY NAME

- When list elements are named, those names can be also used for indexing
- Works both with [ ] and [[ ]] operators
- Indexes, either integer or character (element names) can be passed in a variable
- There is an additional operator that can be used ONLY in conjunction with name LITERALS:
  - `x$b` extracts the element of list `x` named “b” (SAME as `x[["b"]]`) – note the lack of quotation marks in expression `x$b`
  - `x$z` in the example shown on the right asks for *element named “z”* (no such thing, NULL is returned). The *value* stored in variable `z` cannot be used here – one should have used `[[z]]` instead!

```
> x=list(a=1,b=c(5,-3,2),matrix(1:4,ncol=2))
> x[c("a","b")]
$a
[1] 1

$b
[1] 5 -3 2

> x[["b"]]
[1] 5 -3 2
> z="b"
> x[[z]]
[1] 5 -3 2
> x$b
[1] 5 -3 2
> x$z
NULL
```

# NEED FOR “TABLES”

- Think about a typical data table (e.g. an Excel worksheet or relational database table): a table is usually composed of columns of different data types
  - For instance, employee name (character), SSN (integer), date of hire (date), department (integer id or character name/code), pay rate (integer or float), ...
  - Obviously a very ubiquitous data structure. Any relational database supports it (because that's what a relational database is about in the first place) and most languages would have some kind of an extension library – or you would end up writing such class by yourself sooner or later
- Matrix type (in R) would not cut it: all elements must have same type; keep all the data in character mode (the most general data type) and convert to integer/float/date etc as needed? – very inelegant and cumbersome
- List (in R) – sounds like a better solution: each element of a list can store an individual column (a vector), each list element (column) can have its own type, consistency of the type is enforced across each column (so much better than e.g. list of lists in Python, here we have something more similar to list of arrays instead)
  - Still many drawbacks: cannot enforce same length for all columns (a table has  $N$  rows  $\rightarrow$  all columns must be of length  $N$ !)
  - Awkward access: element at row  $i$  column  $j$  is `x[[j]][i]` (yes, at least you can chain access operators in R!), but no easy way to ask for a whole row or a few rows at all!

# DATA FRAMES

- Meet data frame type (this week's assignment heavily relies on them!)
  - Based on a list (in many ways data frame still IS a list as we will see shortly)
  - Fixes the drawbacks mentioned earlier by enforcing rectangular shape and providing matrix-like access operator [ row, column ] (can take vectors of indices – consistent with other R data types)
- R is very much about data analysis: data frames are extremely important and expected by many functions as arguments

Note that the constructor is very similar to that of a list: the column vectors are passed as arguments, with optional names

```
> m2 <- data.frame(age=c(23,43,32),sex=I(c("M","F","M"))); m2
  age sex
1  23  M
2  43  F
3  32  M
> class(m2)
[1] "data.frame"
> mode(m2)
[1] "list"
> m2$age # we can use list operator '$' to access dataframe columns
[1] 23 43 32
> mode(m2$age)
[1] "numeric"
> mode(m2$sex)
[1] "character"
```

We will discuss the “magic” of I() later



# ACCESSING DATA IN DATA FRAMES

- Data frame IS a list of columns (as `mode()` tells us), and the “conventional” list access works for dataframes too
  - `x$column.name` will return the whole column (as a vector!) from the dataframe `x`
  - `x[["column.name"]]` will do exactly the same, and so will `j<-"column.name" ; x[[ j ]]`
  - In addition, matrix like operator `[ i , j ]` will extract a subset of rows/columns
  - All the usual indexing options are available with `[ , ]`: integer indexes, character names (if rows/columns have names set), or logical vectors of TRUE/FALSE “flags”
  - **IMPORTANT:** when using `[ , ]` to select elements (one or many) from a *single* column, the result is a vector; when selecting multiple columns (even if the columns have the same mode and even *when a single row is selected*), the result is always a data frame, never a matrix or a vector!
  - Since dataframe IS a list of columns, `names(x)` will work on dataframes too (and access/set the column names)
  - Additionally, `rownames(x)` and `colnames(x)` can be used to set/retrieve row/column names (just like it works with matrices)
  - Appending an element to a column will expand ALL columns accordingly (filling them with NA)
  - On data frame initialization or column assignment, values may be recycled as usual (if possible), but otherwise an attempt to pass/set columns of different lengths causes an error

# DATA FRAME EXAMPLES

- Continuing with the same data frame m2 defined earlier:

```
# add a new column "weight":
> m2$weight<-c(167,135,202)
> m2
  age sex weight
1  23   M   167
2  43   F   135
3  32   M   202
# multiple cols, still a
# dataframe :
> class(m2[1,c(1,3)])
[1] "data.frame"
> mode(m2[1,c(1,3)])
[1] "list"
# but single column is
# returned as a vector:
> mode(m2[, 3])
[1] "numeric"
> dim(m2)
[1] 3 3
> length(m2)
3
```

```
> m2$healthy<-T; m2 # recycling works
  age sex weight healthy
1  23   M   167     TRUE
2  43   F   135     TRUE
3  32   M   202     TRUE
> m2$does.exercise<-c(T,F)
Error in `<-`(.data.frame`(`*tmp*`, ...
  replacement has 2 rows, data has 3
# a column is a vector, can use as such:
> sum( m2$age > 30 )
[1] 2
# taking a subset of a data frame:
> m2[ m2$age > 30 , ] # see also docs for subset()
  age sex weight healthy
2  43   F   135     TRUE
3  32   M   202     TRUE
> m2[1,5]<-"John D"; m2 # fills missing data with NA
  age sex weight healthy V5
1  23   M   167     TRUE John D
2  43   F   135     TRUE  <NA>
3  32   M   202     TRUE  <NA>
```

What is the  
difference?

# PROGRAMMING IN R: IF-ELSE

- If-else is a conventional flow control operator, works the same way as in any other language
  - The condition must evaluate to a logical vector of *length 1* (the program branching condition is either TRUE or FALSE); using a vector of length  $> 1$  will cause a *warning* and only the first element will be used
  - Condition that evaluates to NA causes an *error*
- Compare to ifelse(): not really flow control operator, but a vectorized data transformation *function*

```
> x <- c(3,5)
# curly braces optional in one-line form:
> if ( any(x > 4) ) "apples" else "oranges"
[1] "apples"
# multiline form - must use curly braces:
> if ( any(x > 4) ) {
  x <- x+3
  print(x)
} else {
  x <- -x*2
  print(x)
}
[1] -6 -8
```

```
> x<-c(3,5,1,7)
# note that recycling occurs below. Each arg of ifelse
# has the same length as the first arg (recycled as
# needed) and for each position in the result (also
# same length as the first arg), the corresponding
# element of either 2nd or 3rd arg is taken, depending
# on whether the condition (first arg) is T or F at
# that position:
> ifelse(x>4,"apples","oranges")
[1] "oranges" "apples" "oranges" "apples"
```

# PROGRAMMING IN R: FUNCTIONS

- Function definition in R creates an *object* that can be *assigned* to a variable, or used as an anonymous function
- Formal parameters allow default values
- In order to pass values for the arguments, function calls can bind formal parameters by name

```
> my.f = function(x,y,option.1=3,option.2="sorted") { # 3rd and 4th args have defaults
  cat("x=",paste(x,collapse=","),"; y=",y,"; option 1=",option.1,
      "; option 2=",option.2,"\n",sep="")
}
> my.f(c(1,2)) # must specify all args that don't have defaults, error otherwise
Error in cat("x=", paste(x, collapse = ","), "; y=", y, "; option 1=", :
  argument "y" is missing, with no default
> my.f(c(1,2),NA) # required args specified, remaining parameters use defaults
x=1,2; y=NA; option 1=3; option 2=sorted
> my.f(c(1,2),NA,"test",2) # "traditional" call: all args passed in correct positions
x=1,2; y=NA; option 1=test; option 2=2
> my.f(c(1,2),NA,option.2="unsorted") # binding by name, specify only what you need
x=1,2; y=NA; option 1=3; option 2=unsorted
# can use binding by name even for all args; the order is irrelevant for named args:
> my.f(y=NA,x=10,option.2="unsorted")
x=10; y=NA; option 1=3; option 2=unsorted
```

# LOADING AND SAVING DATA

- In order to do anything meaningful with data we need to be able to load them first!
- Here is a classical “iris” dataset (described by R.A.Fisher), the data contain sepal and petal characteristics and types (species) of 150 individual iris plants.

```
sepal.length, sepal.width, petal.length, petal.width, class
5.1, 3.5, 1.4, 0.2, Iris-setosa
4.9, 3.0, 1.4, 0.2, Iris-setosa
4.7, 3.2, 1.3, 0.2, Iris-setosa
...
```



- Download the file and load the data (you’ll be doing this twice for this week’s assignment!):

```
> setwd("C:\\Users\\...\\Week 2") # change dir to where YOUR data are located!
> getwd() # see what R's current working dir is
[1] "C:/Users/.../Week 2"
> iris <- read.table("iris.data.txt", header=T, sep=",") # read data in! Can also take full path or even a URL!
> iris[1:3,] # see what we got:
  sepal.length sepal.width petal.length petal.width      class
1          5.1          3.5          1.4          0.2 Iris-setosa
2          4.9          3.0          1.4          0.2 Iris-setosa
3          4.7          3.2          1.3          0.2 Iris-setosa
> write.table(iris, file="C:/Work/iris.data.tsv", sep="\t", col.names=T, row.names=F, quote=F) # write to disk!
```

# FACTORS

- R has yet another basic data type that we ignored so far: factors
  - In statistics, a “factor” is another name for a categorical variable, the one that takes on a finite set of fixed values
  - Thus the ‘factor’ data type is most similar to what in many other languages is called ‘enum’ (enumeration type)
  - Very useful and many functions know how to use factors correctly. NOTE: implementation may result in very unexpected consequences if you are not careful when using factors
  - Dataframes convert *character* vectors (columns) into factors *automatically* – both when explicit constructor `data.frame()` is used and when a data frame is loaded with `read.table` – sometimes this is NOT what we want (e.g. customer’s gender, M/F is a bona fide categorical variable, while their name is probably nothing but just an attribute with no particular statistical significance). If **!** `p.names` is a character vector, we can either use `I()` in the constructor (e.g. `data.frame(patient.name=I(p.name), ...)` or set the optional parameter `stringsAsFactors` in order to prevent the conversion; in `read.table()` we can use `as.is=TRUE` (applies to ALL character columns!)

```
> class(iris)
[1] "data.frame"
# this useful function applies specified function to each element of a vector or a list:
> sapply(iris, mode)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
"numeric"    "numeric"    "numeric"    "numeric"    "numeric"
> sapply(iris, class)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
"numeric"    "numeric"    "numeric"    "numeric"    "factor"
```

# WORKING WITH FACTORS

- A few examples of working with factors:

```
> s<-c("M", "F", "M")
> f<-as.factor(s); f
[1] M F M
Levels: F M
> as.numeric(s)
[1] NA NA NA
Warning message:
NAs introduced by coercion
> as.numeric(f)
[1] 2 1 2
> levels(f)
[1] "F" "M"
> as.character(f)
[1] "M" "F" "M"
> f[3]<-"Z"; f
Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "Z") :
invalid factor level, NAs generated
[1] M F <NA>
```

# NOTE ON THE DATASETS

- There is a large and very useful collection of various datasets online (it's not the only source of data in this world, of course):
  - <http://archive.ics.uci.edu/ml/>
  - Visit the website and just browse the collection to get an idea of what's available
  - We will be using some of these datasets for practice!
  - The iris dataset we just looked at was downloaded from that site (in fact, it is also a standard dataset included with R base distribution, you can just execute `data(iris)` instead of loading manually from a file)
  - note that with the data representation chosen on the above website, the data file has *only* data; the attribute (column, or variable) meanings and names are defined in a separate file; we chose to edit the downloaded data file by adding column names – slightly “conditioning” your data files is something you might want to consider in general when working with a dataset, especially repeatedly. Note that *for the homework* you are asked to **not** edit the data files but to set column names programmatically after the data are loaded!
  - Always **LOOK AT YOUR DATA** before loading them (header/no header? Comma or tab-separated? Are text values enclosed in quotation marks? How missing values are represented?)



# OUTLINE

- More R
  - Lists, dataframes, factors
  - Conditionals and loops
  - Functions
  - Loading data
  - Factors
- Exploratory analysis
  - Boxplots, scatterplots
- Statistical learning: what is it about?
  - Statistical modes
  - Model bias
  - Prediction vs inference
  - Prediction accuracy

# SUMMARY STATISTICS

- ALWAYS explore your data and LOOK at them before jumping into “statistical analysis” let alone “statistical learning”
  - Variables may have funny distributions
  - Data may contain outliers
  - What are the noise levels?
  - Are the (categorical) data well balanced (e.g. does your dataset have data for 376 females and 2 males)?
  - What type of a trend is present, if any?
  - Is there a batch effect?
- We have discussed mean and variance, but median and interquartile distance may be also useful
  - Convey information similar to mean/variance: the “location” and the “width” of the distribution
  - Non-parametric statistics, much more robust against outliers
- Let us sort all  $N$  observations in ascending order
  - Median = the observation in the middle of the list (at position  $N/2$ )
  - $Z\%$  quantile: a value  $X$  such that  $Z\%$  of data are below  $X$  (i.e.  $X$  is the value at position  $N*Z/100$  in the sorted list) [thus median is simply the 50% quantile]
  - 25% and 75% quantiles are also called  $Q1$  and  $Q3$  quartiles (naturally), *interquartile range* or ***IQR*** is  $Q3-Q1$

# BOXPLOT

- It is difficult to look at long lists of numbers:

```
# summary can be applied to a vector; when applied to a dataframe, it is applied  
# to each column separately
```

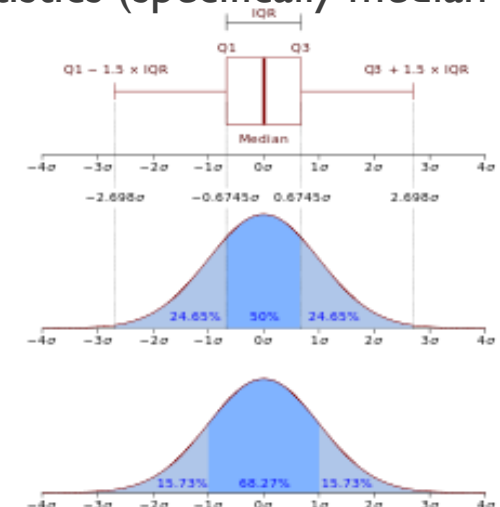
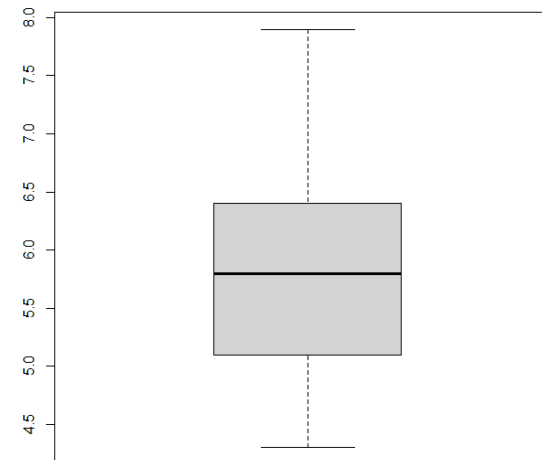
```
> summary(iris)
```

| Sepal.Length  | Sepal.Width   | Petal.Length  | Petal.Width   | Species       |
|---------------|---------------|---------------|---------------|---------------|
| Min. :4.300   | Min. :2.000   | Min. :1.000   | Min. :0.100   | setosa :50    |
| 1st Qu.:5.100 | 1st Qu.:2.800 | 1st Qu.:1.600 | 1st Qu.:0.300 | versicolor:50 |
| Median :5.800 | Median :3.000 | Median :4.350 | Median :1.300 | virginica :50 |
| Mean :5.843   | Mean :3.057   | Mean :3.758   | Mean :1.199   |               |
| 3rd Qu.:6.400 | 3rd Qu.:3.300 | 3rd Qu.:5.100 | 3rd Qu.:1.800 |               |
| Max. :7.900   | Max. :4.400   | Max. :6.900   | Max. :2.500   |               |

- Boxplot is a very convenient (and customary) way of representing the summary statistics (specifically median and IQR) graphically

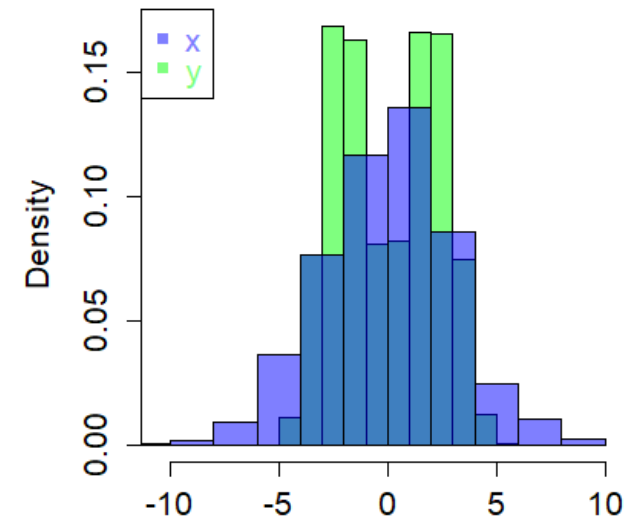
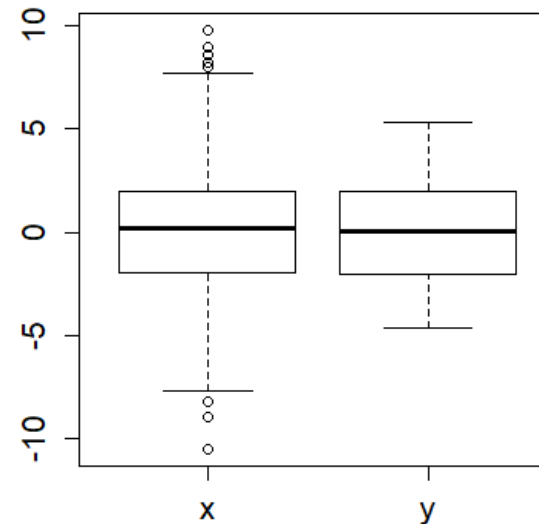
```
> boxplot(iris$Sepal.Length,col="lightgrey")
```

WILL BE USED IN THIS WEEK ASSIGNMENT!



# BOXPLOT VS. HISTOGRAM

```
x <- rnorm(1000,sd=3)
y <- c(rnorm(1000,mean=-2),
      rnorm(1000,mean=2))
old.par <- par(mfrow=c(1,2),ps=16)
boxplot(list(x=x,y=y))
clrTmp <- c(rgb(0,0,1,0.5),rgb(0,1,0,0.5))
hist(y,col=clrTmp[2],freq=FALSE,xlab="",
     xlim=range(c(x,y)),main="")
hist(x,col=clrTmp[1],add=TRUE,freq=FALSE)
legend("topleft",c("x","y"),pch=15,
     col=clrTmp,text.col=clrTmp)
par(old.par)
```



- Of course, similarly looking boxplots do not imply “the same” distributions
- In the above artificially generated example, two samples, one from a bimodal and one from a unimodal distributions, are compared. Although noticeably different when represented as histograms, they look very similar as boxplots
- But then, two random samples drawn from the same distribution, especially when relatively small in size, can also look fairly different when plotted as histograms, while for as long as we look at boxplots, i.e. concentrate on their high-level statistics (median, IQR), they will likely look more similar. ALWAYS be aware of what you are looking at!

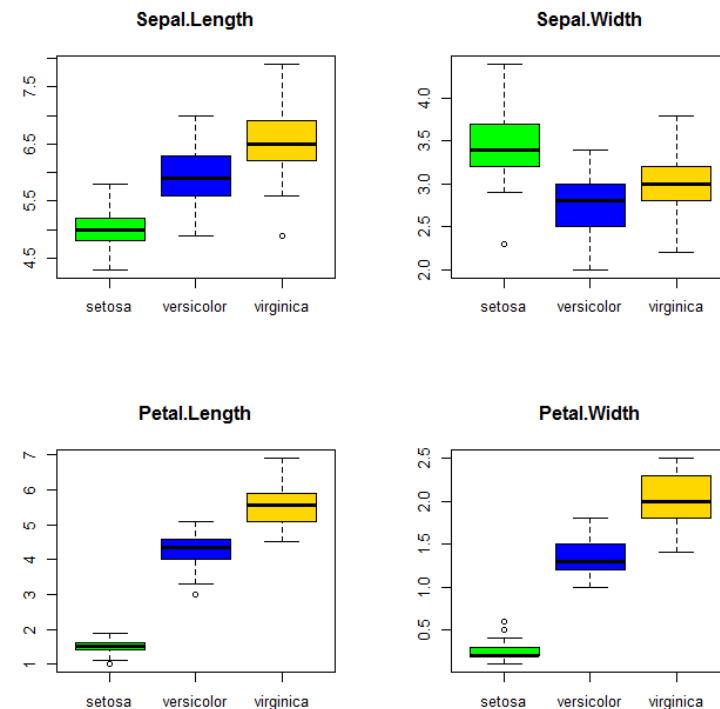
# STRATIFIED (CONDITIONAL) DISTRIBUTIONS

- Boxplots are very useful for comparing different distributions side by side
- Let us stratify our continuous variables (lengths and widths) by the categorical one (species):

```
# set graphical parameters - in this case, split drawing canvas into 4 separate panels,  
# each subsequent plotting command will draw into the next panel (in this case by row).  
# we want to save the old set of graphical parameters if we need to restore them later:  
oldpar <- par(mfrow=c(2,2))  
for ( i in 1:4 ) {  
  # note the "formula interface" in the boxplot call:  
  boxplot(iris[[i]] ~ iris$Species,  
    main=names(iris)[i], col=c("green", "blue", "gold"))  
}  
par(oldpar)
```

- When used in a boxplot call, the syntactic construct  $y \sim x$  means “split the values from vector  $y$  into separate subsets according to the distinct values found at corresponding positions in the factor  $x$ ; draw boxplots of the resulting subsets side by side”

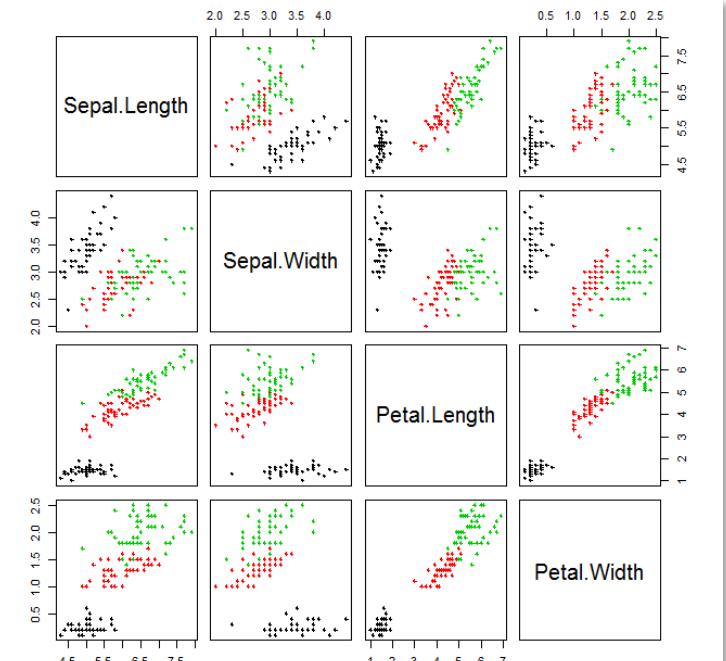
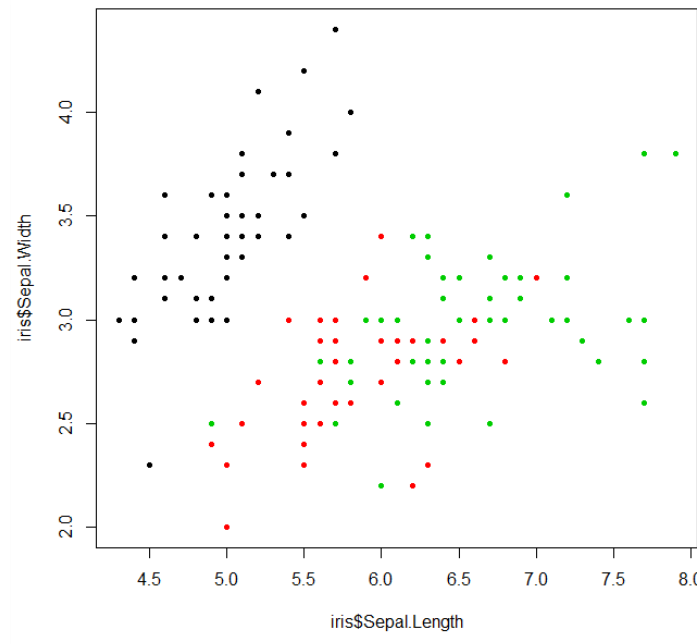
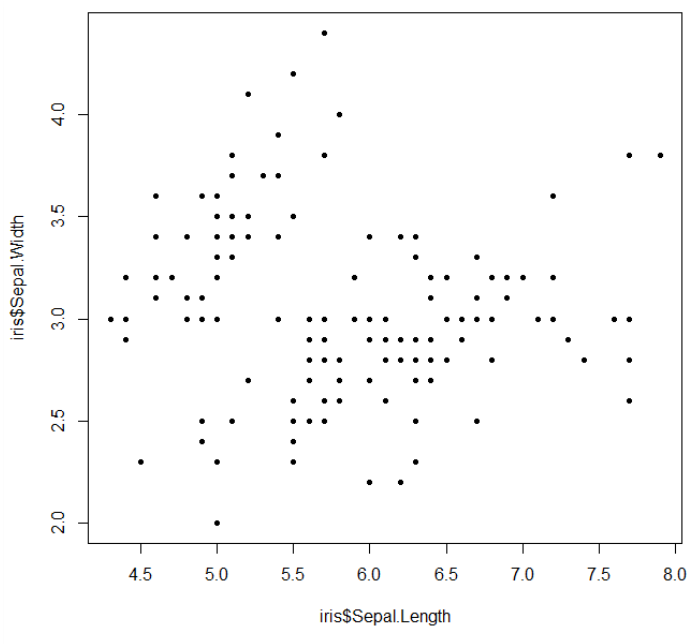
WILL BE USED IN THIS WEEK ASSIGNMENT!



# SCATTERPLOTS

- For continuous variables, it is often useful to generate scatterplots (to see interdependencies/trends, if any)
  - There is also a separate command in R that can generate multiple scatterplots at once, see below

```
plot(iris$Sepal.Length,iris$Sepal.Width,pch=19,cex=0.7)  
plot(iris$Sepal.Length,iris$Sepal.Width,pch=19,cex=0.7,col=iris$Species)  
pairs(iris[1:4],pch=10,cex=0.5,col=iris$Species)
```



# OUTLINE

- More R
  - Lists, dataframes, factors
  - Conditionals and loops
  - Functions
  - Loading data
  - Factors
- Exploratory analysis
  - Boxplots, scatterplots
- Statistical learning: what is it about?
  - Statistical modes
  - Model bias
  - Prediction vs inference
  - Prediction accuracy

# WHAT STATISTICAL LEARNING IS NOT ABOUT...





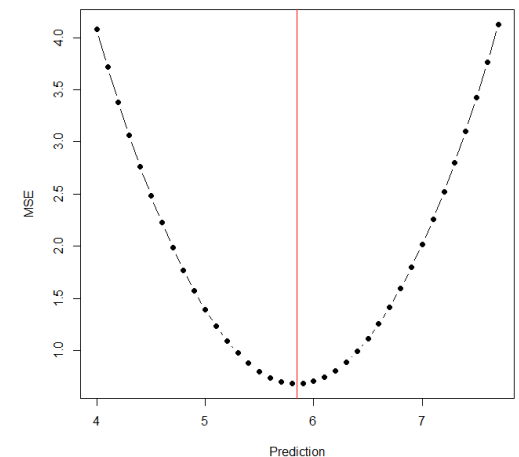
# PREDICTION PROBLEM

- What does it mean to “predict” anything (in statistical sense)?
  - There is a quantity of interest, an *outcome*, *response*, or *dependent variable* (all interchangeable), often denoted as  $Y$  (it is a random variable!)
  - We have some “historical” (previously measured) values (a realization)  $y_1, y_2, \dots, y_k$
  - We want to know what the next measurement is going to be:  
 $Y = [\text{prediction generator}]$
  - Our “prediction generator” gives some value  $\hat{y}$  (we use the “hat” to indicate *estimates* obtained with a model). Ideally, we would like to have  $y - \hat{y} = 0$ , for all measurements, past and, especially, future. Not going to happen, most of the time. We can characterize predictions by their accuracy, i.e. by how far away they are from true measured values. For instance,  $\frac{1}{k} \sum_{i=1}^k (y_i - \hat{y}_i)^2$  - this quantity is known as mean squared error, or MSE.
  - Our goal is to produce “the best” prediction generator. When we use previously measured data to “tune” our generator in a certain way – we are “learning from example”
  - In most (all?) practical settings, we have other variables measured,  $X_1, \dots, X_n$ , and we want to use those to help us with prediction

# PREDICTION AND VARIANCE: SINGLE VARIABLE

- Try to predict the following:
  - Will the sun rise tomorrow (not so fast, what about Vogons)?
  - What is the result of the next coin toss?
- Probability distribution is all we need to make a prediction. What is the best prediction we can make if the distribution  $P(Y)$  is all we know (i.e. we do not measure anything else)?
  - Try predicting the height of the next person to come into this room
  - Try predicting the sepal length of the next iris plant you are going to see
  - Note: we are not playing a lottery, we want the prediction strategy that works best long term i.e. *on average*
  - Let us try different “predictions” for the sepal length and see how they perform in terms of MSE:

```
mse=numeric()
for ( prediction in seq(4,7.7,by=0.1) ) {
  mse=c(mse,sum( (iris$Sepal.Length - prediction)^2 )/150)
}
plot(seq(4,7.7,by=0.1), mse, xlab="Prediction", ylab="MSE", pch=19, type="b")
abline(v=mean(iris$Sepal.Length),col="red")
```

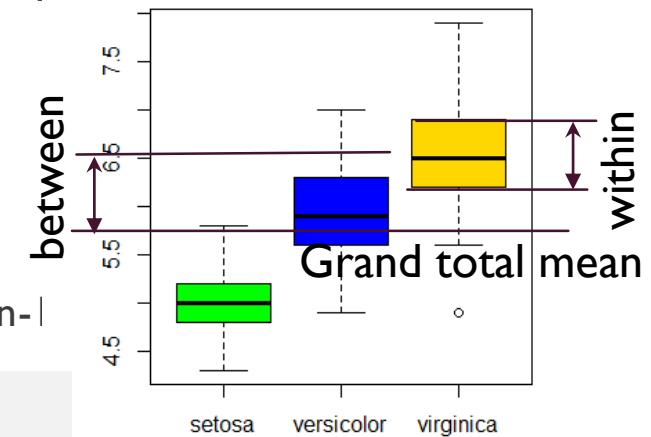


- Note: if we defined the error function as  $\frac{1}{k} \sum_i |y_i - \hat{y}_i|$  the best prediction would be the median rather than the mean!

# PREDICTION AND VARIANCE: EXPLANATORY VARIABLES

- Now let's recollect that the sepal length seem to depend on the species (yes, please note that for the sake of example we are trying to see here if we can predict *sepal length* from species, not the other way around)
- In the presence of an “explanatory variable” we can often predict much better!
  - What our predictions are going to be for each species?
- Let us look at sums of squares in different scenarios (we are using function `var()` here simply for convenience, as variance is by definition just a sum of squared differences from the mean, divided by  $n-1$ )

```
v.total = var(iris$Sepal.Length) # total sum of (observation - mean)^2 (divided by N-1)
levels(iris$Species)
[1] "setosa"      "versicolor" "virginica"
v.setosa = var(iris$Sepal.Length[iris$Species=="setosa"]) # in setosa, vs respective mean
v.versi = var(iris$Sepal.Length[iris$Species=="versicolor"]) # in versicolor
v.virg = var(iris$Sepal.Length[iris$Species=="virginica"]) # ...
m1=mean(iris$Sepal.Length[iris$Species=="setosa"])-mean(iris$Sepal.Length)
m2=mean(iris$Sepal.Length[iris$Species=="versicolor"])-mean(iris$Sepal.Length)
m3=mean(iris$Sepal.Length[iris$Species=="virginica"])-mean(iris$Sepal.Length)
v.total*149
[1] 102.1683
(v.setosa+v.versi+v.virg)*49 + (m1^2+m2^2+m3^2)*50
[1] 102.1683
```



Note: the figure is an illustration only since boxplots show medians/IQRs, not means/variances

We used the fact that  $\text{variance} = \frac{\sum (\text{squared diff from mean})}{(\text{Nobservations}-1)}$

# PREDICTION AND VARIANCE: CONTINUED

- What we observed (can be proven, of course!) is that the sum of squared “errors” in the best possible prediction without explanatory variables (which is just the sample’s grand mean) is equal to the sum of squares of *remaining* errors of predictions made *with* explanatory variable taken into account, PLUS the squared differences between new predictions and the sample mean.
- Variation within each species (“within” term) is still unexplained – even if we know the species, we cannot predict better than just the mean *in that subset*; those are the “remaining errors”
- Variation between different predictions (means of the different subsets) and the sample mean (“between” term) is what we *explained*: indeed, this is what we *predict* based on the species (literally: “when species=setosa, then sepal length is this, and when species=versicolor, then sepal length is that”)!
- Total variation in the dataset (wrt the grand mean) = Unexplained (remaining) variation (around predictions) + Explained (predicted) variation
- Or in other words, prediction = reducing the remaining unexplained variance!
- At the very high level, the problem of statistical learning/prediction is an attempt to answer the questions:
  - What are the “right” variables that allow the best stratification of the outcome (i.e. most accurate predictions with the least remaining, unexplained variance)? [note: “stratification” is used loosely here, as the predictor variables can be continuous!]
  - What is the best way to *find* that stratification?

# STATISTICAL LEARNING

- Previously, we have written the “idea” of prediction down as  $Y = [\text{prediction generator}]$
- More formal way is to write  $Y = f(X) + \varepsilon$ 
  - Here  $X$  are all the explanatory variables we are considering,  $X_1, \dots, X_n$
  - $f(X)$  is the “prediction generator” – it does not necessarily have a closed functional form (e.g.  $f(x) = Ax + B$ ), but of course it should be computable in some way, e.g. by a computer program. For each measurement (realization)  $x_1, \dots, x_n$  of the random processes  $X$ , we can substitute those values into the prediction generator and we get back the corresponding predicted value  $\hat{y}$ .
  - The predicted value is not going to be always precisely correct – there will be *unexplained* variance remaining, which is described by the noise term  $\varepsilon$ .
  - Note that we call “noise” everything that our model does not explain: it can be “true” random noise (as in nature itself throwing a proverbial dice) or it can be some effect that perhaps *could* be explained if we knew (or cared) to include additional variables into our model! Return to our Sepal Length example: if you examine the pairwise scatterplots carefully, you could observe that *within* each species sepal length still correlates pretty well with sepal width. So we could improve our model further and explain a bigger chunk of the total variance in the observations of the sepal length (if that’s what we were aiming at predicting)! But for as long as we don’t include sepal width, all the remaining, “within” variance in our boxplots is just that: “noise”.

# PREDICTION VS INFERENCE

- In prediction problem we are interested in predicting outcome in new cases
  - Stock price
  - Patient survival
- In inference problem we are more interested in describing the data at hand
  - Which parameters are important?
  - Which parameters have positive/negative effect on the outcome?
- In general, there is no clear distinction between models that can be used for prediction or inference, but the properties of different models may make them more or less desirable in different settings
  - For instance, extremely complex model that does not allow for clear interpretation of the dependencies or relative importance of the variables might be something we are more willing to adopt for prediction but not for inference

# MODEL BIAS AND VARIANCE

- As we have been discussing the prediction generator, or the *model* (which is the proper term)  $f(X)$ , we ignored the fact that there is a difference between “true” model (or the “best possible one”) and the one we explicitly use to explain the observations. This difference is called the *model bias*
  - Again, the true model may or may not have any particular closed functional form. We refer to the best possible model here in statistical sense: the one that gives the most accurate predictions, in the presence of noise
- You will also hear a lot about model bias-variance tradeoff (and/or decomposition)
- We will look closer at bias-variance decomposition and tradeoff later in the course
- For now it suffices to say that model variance in this context refers to variability across the fits of the *same* model to *different* training datasets (i.e. different random samples from the same underlying population)
  - Think of the sample mean: this is, in fact, the most trivial “model” (as we just discussed): if we do not include anything else, all we can do is just “train” to predict the average outcome  $Y$ , based on available observations  $y_1, \dots, y_n$
  - As we have already discussed, every time we draw a new sample (“new dataset”), the “learned” sample mean is going to be somewhat different. Those differences are *model variance* (dataset-to-dataset)
  - If in reality there is an explanatory variable  $X$  such that, for instance,  $y_i = a x_i + \epsilon_i$ , but we were oblivious to that fact, then our simple model has a *bias* (the true underlying dependence is different from what we thought it was!)

# OVERFITTING

- A model can perform great on the training dataset, but does it generalize well?
- We can have so many parameters in the model/so few data points that our model starts fitting the noise!
- If prediction is what we are after, we **MUST** evaluate our model on a test dataset – a dataset that has **NOT** been used for training. Accuracy on the *test* dataset is of the greatest importance.
- Data that we are trying to fit a model to are only a *sample* (randomly drawn!). Just like sample mean is a random variable, so is any other quantity that is computed from a *given sample*. Thus, it also includes a model itself (or rather its *parameters*)
  - Models that are too flexible will have high **variance**: they change a lot from one training dataset to the next, i.e. fit noise
- **Bias** is the error due to oversimplification of the model, i.e. the (expected value of the) difference between what the prediction could be if we knew the “true” dependence and the prediction that our model makes.
  - Bias-Variance tradeoff: there is a sweet spot where overall accuracy is optimal
  - Remember that the “true model” may or may not have any particular closed functional form. We refer to it here, somewhat informally, as some “true underlying dependence”.



# SUMMARY

- We have finished the review of basic capabilities of R
  - lists, data frames and factors
  - Flow control, functions
  - Loading and saving data
- Few basic techniques and approaches for preliminary/exploratory data analysis, summary statistics and plots
- “Prediction” interpreted in a sense of expecting the most “likely” outcome (there is a reason for defining the mean of the distribution as *expected value* of  $x$ , or  $E(x)$ )
  - If we have a distribution for outcome  $Y$ , and nothing else, then the best prediction is the mean (if the error metric is the MSE!)
  - When we have additional variables  $X$  measured, we essentially try computing (or estimating)  $P(Y|X)$  (this is *exactly* what the stratification we performed in iris dataset represented: we built the distributions of the sepal length for each species *separately*, i.e. we looked at  $P(\text{Length}|\text{Species})$ !)
  - The difference(s) between the predicted values (at different values of  $X$ ) are the part of total variance in  $Y$  that we *explain*. The remaining variance of  $P(Y|X)$  at any fixed  $X=x_i$  is *unexplained* variance.
- Discussion of statistical modeling: models, their complexity and flexibility, model bias, overfitting (model variance), bias-variance tradeoff.

## ADDITIONAL DETAILS

- Above covers basic necessities to complete this week assignment
- For the remaining time we will cover few more topics closely related to those already discussed

# LISTS: OUT OF BOUNDS ACCESS

- Just like vectors, lists allow out of bounds access without causing an error
  - On access, a NULL will be returned (not an NA!!)
  - On assignment, the list will auto-expand and NULL elements will be inserted as needed.

```
> x=list(1:3);x
[[1]]
[1] 1 2 3

> x$V="new element"; x
[[1]]
[1] 1 2 3

$V
[1] "new element"
```

```
> x[5]=2
> x
[[1]]
[1] 1 2 3

$V
[1] "new element"

[[3]]
NULL

[[4]]
NULL

[[5]]
[1] 2
```

# VARIABLE SCOPING

- The body of if/else, 'while' or 'for' statement does NOT have its own scope (variable first defined inside if/else or a loop is available afterwards)
- Variables defined inside functions (including formal parameters) are local to those functions and will *mask* variables with the same names in the outer scope
- Outer scope ("global") variables are visible inside function body but unmodifiable (sort of)
  - If you try to *modify* a global variable inside a function, this will result in creating a local copy ("copy-on-change")!
  - It is possible to modify a global variable (and even the argument passed to a function!) from inside a function, but you really shouldn't!

```
> y=1:3
> f=function() {
  print(y)
  y[2]=5 # now y is local!
  print(y)
}
> f()
[1] 1 2 3
[1] 1 5 3
> y
[1] 1 2 3
```

```
> y=1:3
> f=function() {
  print(y)
  y[2]<-5 # assign in outer scope
  print(y)
}
> f()
[1] 1 2 3
[1] 1 5 3
> y
[1] 1 5 3
```

# LAZY EVALUATION

- Evaluations in R are performed lazily (only when needed), and that includes *function arguments*
  - Logical expressions are short-circuited (something you most likely have seen before)
  - Function arguments and even *default values* are not evaluated until and unless they are needed at runtime!

```
> x=5
# need to evaluate both expressions:
> ! is.na(x) && x > 0
[1] TRUE
> x=NA
# evaluates only part to the left of &&:
> ! is.na(x) && x > 0
[1] FALSE
# proof: undefined function - no problem!
> ! is.na(x) && some.undefined.fun(x)
[1] FALSE
# Can you explain what happened here:
> F & some.undefined.fun(x)
Error: could not find function
"some.undefined.fun"
```

```
f=function(x,y=sqrt(z)) {
  if ( x == 1 ) print("I need only x")
  else {
    y=y+1; print("I need x and y")
  }
}
> f(1) # y never used, default not evaluated!
[1] "I need only x"
> f(1,sqrt(W)) # y not used, passed arg not evaluated!
[1] "I need only x"
> f(2) # needs y for y=y+1, cannot evaluate default!
Error in f(2) : object 'z' not found
> z=4 # now z is defined..
> f(2) # and the function can evaluate the default for y!
[1] "I need x and y"
> f(2,y=sqrt(W)) # arg is evaluated since y is needed
Error in f(2, y = sqrt(W)) : object 'W' not found
```

# TRADITIONAL LOOPS VS VECTORIZED CODE

- A word on vectorized arithmetic (cannot be overstated): built-in vectorized arithmetics is written in C and is fast; loops driven by the R interpreter itself are slow.

```
> x<-numeric(10000000)
> system.time(x[] <- 1)
  user  system elapsed 
0.03    0.00    0.03 
> system.time(for(i in 1:length(x)) x[i] <- 1)
  user  system elapsed 
10.44    0.02    10.54
```

- R provides mapping functions (the 'apply' family); contrary to popular belief they are not necessarily faster than R loops, but at least they are convenient and fit well into the whole paradigm of vectorized calculations:

```
> y=numeric(length(x))
> system.time(for(i in 1:length(x)) y[i]=x[i]^2)
  user  system elapsed 
16.22    0.00    16.27 
> system.time(y <- apply(x,FUN=function(x) x^2 ))
  user  system elapsed 
18.82    0.06    18.91 
> system.time(y <- x^2)
  user  system elapsed 
0.02    0.00    0.02
```

Why can't we use '='?

```
> m=matrix(1:6,ncol=2)
> m
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
# apply a function to each row (MARGIN=1);
# to apply to each column we'd use MARGIN=2:
> apply(m,MARGIN=1,sum)
[1] 5 7 9
```

# EXTENDING R

- R can be easily extended by writing functions
  - technically, one can write an efficient implementation in C and wrap it as an R function too – well beyond our scope
- The easiest way to manage a collection of functions (which you most likely will end up having if you continue doing data analysis with R!) is:
  - Save your function definitions into a file (e.g. myCS63library.R)
  - Read the code from the R script file in: `> source("myCS63library.R")`
  - R simply reads and executes the “sourced” code line by line, so you may have a bona fide script too, which not only defines functions (function definitions ARE R commands, of course), but executes some other commands, e.g. performs end-to-end computation, draws plots etc
- Third party libraries are released as “packages” – a more sophisticated way (which we will not go deep into, just use)
  - at the basic level, a package is a library that needs to be (a) installed on your system only once and (b) loaded into your R session every time you need it (once per session); as the result, additional functions (and maybe variables/data) from the package will become available
  - R GUI offers packages->install packages.. option with graphical package chooser interface, RStudio has even better GUI

# LOADING PACKAGED LIBRARY/DATA

- When the package is installed, you have to load it into your session, for instance:

```
> library(ggplot2)
```

- Note that you do not need the quotation marks
- Some packages also (or only!) provide datasets. Those can be loaded as in the following example:

```
> data(iris)
> iris[1:3,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4          0.2  setosa
2           4.9         3.0          1.4          0.2  setosa
3           4.7         3.2          1.3          0.2  setosa
```

- Package 'iris' does not need to be loaded, it's part of base R distribution, so we can attach the data right away.
- When dataset is loaded it creates variable 'iris' in the workspace (so the old value we had after downloading the data file manually is now overridden, actually)