R tutorial

Contents

0.0.1 Basic R commands

First function in R will ask you whether you want to save data and then it will close R:

```
q()
```

This will close R without asking you:

```
q("n")
q(save="n")
```

To show `plot` as an example of function, type:

```
a <- 1:100
b <- sin(a)
plot(a, b)
```

This will work:

```
plot(a, b, main="My plot")
```

The R engine knows that first two parameters of the function `plot` are coordinates `x` and `y`, respectively. However, this will not work:

```
plot(a, b, "My plot")
```

because R does not know that `"My plot"` links to the parameter `main` (the main title). It is possible to specify all options:

```
plot(x=a, y=b, main="My plot")
```

Help could be found by:

```
help(plot)
?plot
apropos("svd")
help.search("svd")
```

Example or demo can be found as well:

```
example(image)
demo(graphics)
```

Basic operations:

```
1+1
2-1
3*3
6/3
```

R recognizes integers and floats from the context:

```
5/2
5.0/2.0
```

Power, modulo, integer division:

```
3^3
3**3
5%%2
5%/%2
```

R waits until command finished:

```
1+
1
```

The hash sign `#` is used to add comments (code after `#` is ignored)

```
1+1#+1
```

Spaces are ignored:

```
1+1
1 + 1
1+          1
1            +1
```

Basic constants and fuctions:

```
pi
cos(pi)
sin(pi)
exp(1)
abs(-4)
```

Logarithm is natural by default, decadic and binary are available as well:

```
log(exp(2))
log10(1000)
log2(16)
```

Complex numbers are supported as well:

```
2i
2i*2i
```

One can assign a value to a variable:

```
x <- 20
x
y <- 10
y
x+y
```

Logical and string variables are available as well:

```
x<-FALSE
x
y<-"string"
y
```

Equal works as well in most cases, but use `<-` to be on the safe side.

```
x = 20
y = 10
x+y
```

It is possible to change (e.g. increment) the value of a variable:

```
x <- 10
x <- x + 1
x
```

R recognizes capitals and small letters:

```
a<-1
A<-2
a
A
a+A
```

Vectors can be defined by function `c()`:

```
x <- c(1, 3, 2)
x
```

Vectors with increments of 1 can be easily generated by:

```
x <- 1:10
x
```

It can be used in the oposite way:

```
x<-10:1
x
```

It is possible to print individual items from the vector:

```
x <- c(1,5,2,3,4,7)
x
x[1]
x[2]
x[3:6]
x[c(1,3)]
```

Another way how to make a vector:

```
seq(from=6, to=21, by=2)
rep((1:4), times=2)
rep((1:4), each=2)
```

It is possible to add, subtract, multiply or divide a vector by a number:

```
x<-1:5
x*2.5
x/2.5
x+2.5
x-2.5
```

It is possible to sum or multiply two vectors element-wise:

```
x<-c(1,3,2)
y<-4:6
x+y
x*y
```

To get a dot product you need to type:

```
x%*%y
```

You can apply a function item-wise on a vector:

```
x<-1:4
exp(x)
```

Matrix can be created by function `matrix` with two possible `byrow` option (`FALSE` is the default):

```
x<-matrix(1:12, ncol=3)
x
x<-matrix(1:12, ncol=3, byrow=TRUE)
x
```

Matrix can be transposes:

```
t(x)
```

Matrix can be created by combining and adding columns or rows:

```
x<-1:4
y<-c(3,2,6,5)
rbind(x, y)
cbind(x, y)
```

Matrix elements can be accessed by indexes in square brackets (`[1,]` is for the first row, `[,1]` for the first column):

```
x<-1:4
y<-c(3,2,6,5)
xy <- cbind(x, y)
xy
xy[1,]
xy[1,1]
xy[1,2]
xy[,1]
xy[,2]
```

Special object for data analysis in R is `data.frame`:

```
x<-1:4
y<-c(3,2,6,5)
mydata <- data.frame(x,y)
mydata
mydata[1]
mydata[2]
mydata$x
mydata[1,1]
mydata[2,1]
```

```
mydata[2,]
```

Logical operations can such as negation, or and and can be used in operations with `data.frame`, lets show some:

```
x<-TRUE
!x
y<-FALSE
x|x
x|y
y|y
y&y
x&y
x&x
1<2
1>2
1==1
1==2
```

Loops are not as important R as in classical programming languages, but it is possible to use them:

```
for(i in 1:3) {
  print(i)
}
```

In R it is possible to define a new function:

```
sinpluscos <- function(x) {
 y<-sin(x)+cos(x)
 return(y)
}
sin(1)+cos(1)
sinpluscos(1)
```

R contains many pre-defined datasets:

```
faithful
data()
```

Beside standard R functions and datasets it is possible to install new packages from R repositories:

```
installed.packages()
install.packages("igraph")
library(igraph)
```

That's all, lets try to read some data in the next lesson and then we will analyze them.

0.0.1.1 Tips and tricks

- you don't remember exactly the command (e.g. you are not sure whether it is `len` or `length`)? Just type the beginning of the command and double-click tabulator. You can use it also for parameters of a function.
- if the command is too long you can separate it on two or more lines (R waits until the command is closed).
- it is better not to use trivial names for variables because you can overwrite some existing variables.
- to generate a vector with constant increment (e.g. 2, 4, 6 ... 10):

```
x<-2*1:5
```

- you can reverse the order of a vector by:

```
x<-2*1:5
x[5:1]
```

0.0.2 Reading and writing files in R

You can read data from unformated text files by function `read.table`:

```
mydata<-read.table("https://raw.githubusercontent.com/spiwokv/Rtutorial/master/data/mydata.txt")
mydata
```

Alternatively you can download the file to a working directory of your computer and open it:

```
mydata<-read.table("mydata.txt")
mydata
```

If you don't know which directory is the working one, you can type:

```
getwd()
```

or you can change it by:

```
setwd("C:/Documents")
```

The function `read.table` can be modified by the parameter `header`, it indicates that the first line of the file contains names of columns. Separators can be changed by parameter `sep`. By default, `sep` is set to `""`, i.e. one or multiple spaces can act as separators. Special separators such as tabulator can be defined by regular expressions, e.g. `sep="\t"`.

It is also possible to control quoting characters, strings which are to be interpreted as `NA` (not available) values or comment character.

There are special functions to read formated files, such as `read.csv`, `read.csv2`, `read.delim`, `read.delim2`, `read.fwf` and `read.ftable`. It is also possible to read data from Microsoft Excel, databases (MySQL, SQLite, Oracle, Microsoft SQL Server and others) XML, JSON and other files.

Data can be written by the function `write.table`:

```
mydata<-read.table("https://raw.githubusercontent.com/spiwokv/Rtutorial/master/data/mydata.txt")
mydata
write.table(mydata, "mydata2.txt")
```

R also contains sample data for demonstration of functions. You can see the list by typing:

```
data()
```

For example `faithful` dataset contains waiting times between eruptions and the duration of eruptions of the Old Faithful geyser in Yellowstone National Park (Wyoming, USA).

0.0.2.1 Tips and tricks

- data in a file may be large and it is not useful to print it when you want to check whether data was read correctly. You can use `head` function instead to print the header with the first six lines.

```
head(faithful)
```

Analogously you can use the `tail` function. ### Analyzing data in R

We can show how to extract and manipulate data from a dataset. Let us create a simple "dataset" containing names:

```
jmena <- c("Karina","Radmila","Diana","Dalimil","Melichar","Vilma","Cestmir","Vladan","Bretislav")
```

It is possible to iterate this vector of names by typing square brackets with element ID (R counts from 1, not from 0). To get the first name type:

```
jmena[1]
```

(returns Karina). You can evaluate some expression within the square brackets, e.g.:

```
jmena[1+1]
```

returns Radmila, not Karina Karina. You can use colon operator:

```
jmena[1:3]
```

This returns first three names. You can use more complicated operation:

```
jmena[1:3*2]
```

This returns element number 2, 4 and 6. Negative indexes can be used to remove some element:

```
jmena[-1]
```

(returns all elements except the first one). The last element can be obtained by `jmena[9]` if you know the number of elements or `jmena[length(jmena)]` in case you are lazy to count them. To inverse the order of elements you can use the colon operator as `jmena[9:1]` or `jmena[length(jmena):1]`.

Let us move to more complicated dataset. You can load the results of municipal elections 2015 in Prague:

```
volby <- read.table("https://raw.githubusercontent.com/spiwokv/Rtutorial/master/data/volby2013praha.
                    sep=";", header=T, dec=",")
```

Parameter `sep=";"` indicates that individual items are separated by a semicolon. The option `header=T` indicates that the first line contains names of columns. In fact the first line contains the remark, because it starts with `#`. The option `dec=","` tells us that the Czech decimal is used. For users with troubles with Czech characters you can replace "volby2013praha.txt" by "volby2013prahaASCII.txt".

For the first inspection you can type:

```
head(volby)
```

This prints the first 10 lines. You can chose different number by `n=2` option. Similarly you can print last lines by function `tail`.

Function:

```
dim(volby)
```

prints the dimension (number of lines and colums). You can get these values separately for the number of lines and columns by functions `nrow` and `ncol`, respectively.

You can iterate on the lines and columns similarly to the previoius example. For example, you can select the first candidate in the alphabetic order as:

```
volby[1,]
```

Columns can be selected by:

```
volby[,4]
```

This prints names of candidates.

The function:

```
names(volby)
```

prints names of columns, such as party number and name, candidate's name and age etc.

As an alternative to `volby[,4]` you can use `$` operator followed by the name of the column:

```
volby$jmeno
```

("jmeno" = "name" in Czech).

The function `levels` determines levels of a vector. For example, if you type:

```
volby[,2]
```

it will print a vector with 680 items (one for each candidate) with political parties of candidates in the alphabet order of their family names. If you place this into the function `levels` it will print each party only once:

```
levels(volby[,2])
```

By function `nlevels` you can get the number of political parties. Function `table` will print a table with numbers of candidates per party:

```
table(volby[,2])
```

In order to print only the lines containing candidates of "Piráti" you can use following expressions: The expression `volby[,2]` will print parties in the alphabet order of names of candidates. You can extend it by `volby[,2]=="Piráti"`. This will return the series of `TRUE` and `FALSE` values in the same order. For example, first 16 candidates in alphabet were not Pirates, so first 16 values are `FALSE`. The candidate number 17 is Pirate, so output number 17 is `TRUE`. You can apply `sum` function on the output. This function counts `TRUE` as 1 and `FALSE` as 0.

If you insert the previous expression `volby[,2]=="Piráti"` into the square brackets of `volby[]` you can select lines containing Pirates:

```
volby[volby[,2]=="Piráti",]
```

The square brackets contain comma inside, because we select lines or columns. The expression `volby[,2]=="Piráti"` is in front of the comma because we select lines. Lines with `TRUE` as the output of `volby[,2]=="Piráti"` are printed, others are not printed.

Let us look at numbers of votes in the column number 8. We can check the range by function `range`:

```
range(volby[,8])
```

This shows that the least successful candidate was not voted at all, the most successful got 37794 votes. You can print all votes sorted by the function `sort`:

```
sort(volby[,8])
```

To get the reverse order use option:

```
sort(volby[,8], decreasing=TRUE)
```

You may be interested who scored the best and worst in elections. You can use function `order`. This function prints the index of the lowest value, the index of the second lowest value and so forth. The expression:

```
volby[order(volby[,8]),]
```

will print the table sorted by the number of votes from the lowest to highest. You can revert the order by option `decreasing=T` in the order function.

Finally, we are interested in number of votes for each party. This can be obtained manually, party by party as:

```
sum(volby[volby[,2]=="Piráti",8])
```

and so forth for each party. As an alternative you can use function `aggregate`:

```
aggregate(x=volby[,8], by=list(volby[,2]), FUN=sum)
```

The function `list` is used because votes can be aggregated by multiple factors. Instead of function `sum` you can use other functions, for example average age of each party can be printed by:

```
aggregate(x=volby[,5], by=list(volby[,2]), FUN=mean)
```

You can plot numbers of votes as a pie chart or bar plot:

```
vysledky<-aggregate(x=volby[,8], by=list(volby[, 2]), FUN=sum)
pie(vysledky[,2], labels=vysledky[,1])
barplot(vysledky[,2], names.arg=vysledky[,1])
```

0.0.2.2 Tips and tricks

There is a family of "apply" functions. To calculate a sum for each row of an array or matrix use:

```
apply(myarray, 1, FUN=sum)
```

If you want to calculate the same for columns replace `1` by `2`. You can use any other function with a single input, or even a user defined function defined by `function()`. For example you can count positive values per column as:

```
apply(myarray, 2, function(x) length(x[x>0])
```

There are specialized packages for data analysis such as "dplyr". It uses a special "pipe" operator (`%>%`). The output of the operation before the pipe is used as an input of the operation after the pipe. Special functions `mutate`, `select`, `filter`, `summarise`, `arrange` and others are used in dplyr. You can replace the `aggregate` function from the previous example as:

```
library(dplyr)
ifile %>% group_by(strana) %>% summarise(abs=sum(abs))
```

Another useful package for data analysis is "TidyR". Both dplyr and TidyR are from a tidyverse package of packages for data analysis. TidyR uses functions `gather()`, `spread()`, `separate()`, `extract()` and others to reshape data from untidy to tidy datasets.

0.0.3 Plotting in R

The basic plotting function in R is `plot`:

```
x<-1:1000/100
y<-sin(x)
plot(x,y)
```

You can switch between points (default), lines (`type="l"`), both, histogram-like, steps, none and others:

```
plot( x, y, type="l")
```

The function `plot` always plots a new plot. If you want to add new lines or points into the existing plot use `lines` and `points`. Simply open new plot by function `plot` and then use `lines` and `points` without closing the plot:

```
plot(x,y, type="l")
lines(x,cos(x))
points(x,0.5*cos(x))
```

Sometimes it is useful to create an empty plot canvas by `plot` with `type="n"` and add lines and points.

The function `plot` has many additional parameters:

```
plot( x, y, main="parametr main", sub="mysub",
     xlab="myxlab", ylab="myylab", asp=1)
```

You can change colors of point and lines, shapes of points etc.:

```
x<-1:10
plot(x, sin(x), pch=21, col="red", bg="blue", cex=2, lwd=2)
lines(x, sin(x), col="green", lwd=4)
```

Range of the horizontal and vertical axis can be controlled by `xlim` and `ylim`:

```
plot(1:10, xlim=c(0,100), ylim=c(-20,20)))
```

The function plot can be applied not only to a pair of vectors, but also to a single vector, `data.frame` and other objects. Lets try on `data.frame`:

```
x<-c(1,2,3,1,2,1)
y<-1:6
z<-6:1
xyz<-data.frame(x,y,z)
plot(xyz)
```

The function `plot` can be used to other objects as will be shown later.

Using function `text` it is possible to add short strings to points with coordinates x and y:

```
x<-1:4*2
y<-sin(x)
pointnames<-c("first", "second", "third", "fourth")
plot(x,y)
text(x, y, labels=pointnames)
```

R can make pie charts:

```
x<-c(1,1,2,3,2)
nam<-c("first", "second", "third", "fourth", "fiveth")
pie(x, labels=nam)
```

Barplots can be drawn using function barplot:

```
barplot(c(10.1,8.1,9.5,8.3), names.arg=c(10.1,8.1,9.5,8.3))
```

Histograms can be plotted by `hist` with breaks controllable by `breaks` parameter:

```
hist(faithful$waiting, breaks=20)
```

Tukey's boxplots can be ploted by `boxplot` function:

```
x<-c(1.2,2.2,1.3,4.4,3.0,2.2,2.5,2.6)
y<-c(3.3,2.3,1.8,5.5,7.7,7.3,1.9,4.7)
boxplot(x, y)
```

Three-dimensional plots can be presented using image, contour or persp function:

```
x<-0:100
y<-0:100
gauss<-exp(-outer((x-50)**2/200,(y-50)**2/200,"+"))
image(x, y, gauss, col=heat.colors(100))
contour(x, y, gauss, levels=0:10/10, add=TRUE)
persp(x, y, gauss, col="red", theta=30, phi=30, shade=0.75, ltheta=100)
```

Nice 3D plots can be made by the `wireframe` function from the `lattice` library:

```
library(lattice)
wireframe(gauss, shade=TRUE,light.source = c(10,0,10))
```

The shape of plots can be modified by `par` function invoked before the function `plot` or other plotting functions. As an example we can show plotting of four plots on one canvas:

```
par(mfrow=c(2,2))
x<-1:100/10
plot(x, sin(x))
plot(x, cos(x))
plot(x, tan(x))
plot(x, atan(x))
```

R can use a wide range of colors. Pre-defined colors can be shown by functions `colors`:

```
colors()
barplot(1:6, col="sienna")
```

If colors are supplied as a vector, they alternate as shown bellow:

```
barplot(1:6, col=c("sienna", "steelblue", "olivedrab",
                   "navy", "whitesmoke", "whitesmoke"))
barplot(1:6, col=c("sienna", "steelblue"))
```

Shades of gray can be used by the function `gray`:

```
x<-1:50/50
gray(x)
barplot(x, col=gray(x))
```

Colors can be also mixed from red, green and blue by the function `rgb`:

```
x<-0:10/10
rgb(1,1,x)
barplot(x, col=rgb(1,1,x))
```

You can try attractive palettes such as `rainbow`, `heat.colors`, `terrain.colors`, `topo.colors` and `cm.colors`.

Plots can be saved in many bitmap and vector graphical formats by functions `png`, `jpeg`, `pdf`, `svg` or `ps`. After invoking this function with file name as the argument no plot is shown. Instead it is saved to file. This property can be stopped by function `dev.off()`:

```
png("plot.png")
barplot(1:6)
dev.off()
```

The plot is saved into working directory (see funcitons `getwd` and `setwd`).

R together with its packages makes it possible to plot graphs (in the sense of graph theory), heatmaps, word clouds, geographical maps and other special plot types.

0.0.3.1 Tips and tricks

- i often use function rainbow without the violet color:

```
barplot(1:20, col=rainbow(27)[1:20])
```

- high-resolution bitmap plots can be made in vector format and then converted to bitmap using your favorit graphical software
- alternatively, it is possible to use functions for bitmap plotting (e.g. `png`) with following modification:

```
x<-0:100
y<-0:100
png("plot.png", height=8, width=8, units='cm', res=600, pointsize=6)
gauss<-exp(-outer((x-50)**2/400,(y-50)**2/400,"+"))
image(x, y, gauss, col=heat.colors(100), axes=F)
contour(x, y, gauss, levels=0:10/10, add=TRUE, lwd=2, labcex=1.2)
axis(1, lwd=2)
axis(2, lwd=2)
box(lwd=2)
dev.off()
```

This plots the plot in doubled size. In order to further increase the size it is possible to multiply `width`, `height` and `pointsize` in `png`. However, it keeps the same widths of lines and other parameters. To fix this, avoid plotting axes by function `image` (`axes=F`) and instead plot wide axes and box separately. It can be easily modified for other plotting functions.

- to make a movie, use the output file name with regular expression and a loop:

```
png("plot%03d.png")
x<-0:100
y<-0:100
for(i in 25:75) {
  gauss<-exp(-outer((x-i)**2/400,(y-i)**2/400,"+"))
  image(x, y, gauss, col=heat.colors(100))
  contour(x, y, gauss, levels=0:10/10, add=TRUE)
}
dev.off()
```

You can then use some video software (e.g. mencoder from Mplayer) to make a movie.

- a nice and popular plotting library from the "tidyverse" family is "ggplot2".

0.0.4 Random numbers in R

R can generate random numbers with different distributions. It is possible to generate ten random number with normal distribution with mean set to 20 and standard deviation set to 2 (default values are 0 and 1, respectively):

```
x<-rnorm(10, mean=20, sd=2)
x
mean(x)
sd(x)
```

The true mean and standard deviation are not exactly equal to pre-set values, but you can try with larger sets:

```
x<-rnorm(10, mean=20, sd=2)
mean(x)
sd(x)
hist(x, br=20, xlim=c(10,30), col="gray")
x<-rnorm(100, mean=20, sd=2)
mean(x)
sd(x)
hist(x, br=20, xlim=c(10,30), col="gray")
x<-rnorm(1000, mean=20, sd=2)
mean(x)
sd(x)
hist(x, br=20, xlim=c(10,30), col="gray")
x<-rnorm(10000, mean=20, sd=2)
mean(x)
sd(x)
hist(x, br=20, xlim=c(10,30), col="gray")
```

For normal distribution you can also calculate density by `dnorm`, distribution function by `pnorm` and quantile function by `qnorm`:

```
x<-0:400/10
plot(x, dnorm(x, mean=20, sd=2), type="l")
plot(x, pnorm(x, mean=20, sd=2), type="l")
y<-1:99/100
plot(y, qnorm(y, mean=20, sd=2), type="l")
```

The function `pnorm` is an integral of `dnorm` as you can see:

```
x<-0.1*0:230
sum(0.1*dnorm(x, mean=20, sd=2))
pnorm(23, mean=20, sd=2)
```

The function `qnorm` is an inverse function of `pnorm`:

```
y<-1:99/100
plot(y, qnorm(y, mean=20, sd=2), type="l")
x<-qnorm(y, mean=20, sd=2)
points(pnorm(x, mean=20, sd=2), x)
```

There are similar functions for other distributions such as chi-squared distribution (`dchisq`, `pchisq`, `qchisq` and `rchisq`), t-distribution (`dt`, `pt`, `qt` and `rt`), F-distribution (`df`, `pf`, `qf` and `rf`) and many others.

0.0.4.1 Tips and tricks

- you can set seed if you want to generate same random numbers:

```
set.seed(666)
rnorm(5)
rnorm(5)
set.seed(666)
rnorm(5)
```

0.0.5 Univariate descriptive statistics in R

Lets create a sample with mean and standard deviation set to 20 and 2, respectively:

```
x<-rnorm(10, mean=20, sd=2)
```

Basic measures of descriptive statistics, namely minimum, lower quartile, median, mean, upper quartile and maximum, can be obtained by function summary:

```
summary(x)
```

These values can be accessed by special functions:

```
min(x)
quantile(x, probs=0.25)
median(x)
mean(x)
quantile(x, probs=0.75)
max(x)
```

Another useful function can be `range`.

Important plot of univariate descriptive statistics is Tukey's box plot. It plots a box with bottom and top at lower and upper quartile (exactly, at values nearest to lower and upper quartile). The horizontal line is located at median. The whiskers start from the bottom and top of the box. Each whisker goes to maximum/minimum, but it is not longer than 1.5-times of interquartile range. It some point is more distant it is depicted as a point. See bellow:

```
x1<-rnorm(10, mean=20, sd=2)
x2<-rnorm(10, mean=20, sd=2)
x3<-rnorm(10, mean=20, sd=2)
x4<-rnorm(10, mean=20, sd=2)
boxplot(x1, x2, x3, x4)
points(1:4, c(min(x1),min(x2),min(x3),min(x4)), col="red")
points(1:4, c(max(x1),max(x2),max(x3),max(x4)), col="red")
points(1:4, c(median(x1),median(x2),median(x3),median(x4)), col="red")
points(1:4, c(quantile(x1,0.25),quantile(x2,0.25),quantile(x3,0.25),quantile(x4,0.25)), col="red")
points(1:4, c(quantile(x1,0.75),quantile(x2,0.75),quantile(x3,0.75),quantile(x4,0.75)), col="red")
```

Mean estimate can be calculated by function `mean`, or manually:

```
mean(x)
sum(x)/length(x)
```

Variance estimate can be calculated by function `var`, or manually:

```
var(x)
sum((x-mean(x))**2/(length(x)-1))
```

Standard deviation estimate can be calculated by function `sd`, or manually:

```
sd(x)
sqrt(var(x))
sqrt(sum((x-mean(x))**2/(length(x)-1)))
```

Standard error of the mean does not have own function in (basic) R, so we can calculate it manually:

```
sd(x)/sqrt(length(x))
```

0.0.5.1 Tips and tricks

- it is possible to index the function `summary`, e.g. to get minimum by index 1:

```
summary(x)[1]
```

It is not really useful for this function, but you can use it later for other functions.

0.0.6 Confidence intervals in R

Confidence intervals can be calculated in R, for example, as mean +/- s.e.m. multiplied by quantile of t-distribution.

```
x<-rnorm(10, mean=20, sd=2)
sem<-sd(x)/sqrt(length(x))
mean(x)+sem*qt(p=c(0.025,0.975), df=(length(x)-1))
```

The function `qt(p=c(0.025,0.975), df=(length(x)-1))` returns quantile of t-distribution for p=0.025 and 0.975, i.e. for 95 % probability. For 90 % use `p=c(0.05,0.95)`, for 99 % use `p=c(0.005,0.995)` etc.

If you generate 100 random samples (each with 10 items) with mean set to 20 and standard deviation set to 2, you should expect that 95 samples will contain 20 in the confidence interval and 5 will not. Let's try:

```
good<-0
for(i in 1:100) {
  x<-rnorm(10, mean=20, sd=2)
  sem<-sd(x)/sqrt(length(x))
  ci<-mean(x)+sem*qt(p=c(0.025,0.975), df=(length(x)-1))
  if((ci[1]<20)&&(ci[2]>20)) {
    good<-good+1
  }
}
good
```

I obtained 97, close to expected 95.

0.0.6.1 Tips and tricks

- confidence interval can be obtained more easily by `t.test` as will be shown later:

```
x<-rnorm(10, mean=20, sd=2)
t.test(x)$conf.int
t.test(x)$conf.int[1:2]
```

```
sem<-sd(x)/sqrt(length(x))
mean(x)+sem*qt(p=c(0.025,0.975), df=(length(x)-1))
```

0.0.7 One-sample t-test in R

Confidence interval and one-sample t-test are two sides of the same coin. Let us calculate 95 % confidence interval for a sample generated by function `rnorm`:

```
x<-rnorm(10, mean=20, sd=2)
sem<-sd(x)/sqrt(length(x))
mean(x)+sem*qt(p=c(0.025,0.975), df=(length(x)-1))
```

We can make a two-tailed t-test (at the significance level of 5 %) with the null hypothesis that the mean of x is equal to 20. The null hypothesis is rejected if 20 is outside the confidence interval. You can replace `p=c(0.025,0.975)` by `p=c(0.005, 0.995)` for the significance level 1 % etc.

Another option is to calculate criterion R and compare it with corresponding quantile of t-distribution:

```
R<-abs(mean(x)-20)*sqrt(length(x))/sd(x)
R
qt(p=0.975, df=(length(x)-1))
```

The null hypothesis is rejected if R is bigger than the quantile of t-distribution.

The most convinient t-test option is to use the function `t.test`:

```
t.test(x, mu=20)
```

The null hypothesis is rejected if p-value is lower than 0.05 (or 0.01 for the significance level of 1 %). The function `t.test` also provides the criterion R (called t), degrees of freedom, confidence interval and mean.

One-tailed t-test can be done similarly:

```
mean(x)+sem*qt(p=c(0,0.95), df=(length(x)-1))
t.test(x, mu=20, alternative="less")
mean(x)+sem*qt(p=c(0.05,1), df=(length(x)-1))
t.test(x, mu=20, alternative="greater")
```

0.0.7.1 Tips and tricks

- you can iterate on the results of the function t.test:

```
tt<-t.test(x, mu=20)
tt
tt[3]
```

- you can change the significance level for the confidence interval by parameter `conf.level`:

```
t.test(x, mu=20)
t.test(x, mu=20, conf.level=0.99)
t.test(x, mu=20, conf.level=0.999)
```

0.0.8 Two-sample t-test in R

Let us skip a "manual" version of the t-test and proceed directly to the function `t.test`. There are two variants of two-sample t-test, one for equal variances and one for unequal variances. First let us test whether the variances are equal:

```
healthy<-rnorm(10, mean=12.3, sd=3.3)
healthy
sick<-rnorm(10, mean=8.5, sd=3.3)
```

```
sick
var.test(healthy, sick)
```

The null hypothesis of the `var.test` is that variance of both samples are equal. We can reject the null hypothesis when p-value is lower than 0.05 (for the significance level of 5 %).

The null hypothesis of t-test is that both means are equal. We can reject the null hypothesis when p-value is lower than 0.05 (for the significance level of 5 %). For samples with equal variances we will use t-test with `var.equal=TRUE`. For samples with unequal variances we will use t-test with `var.equal=FALSE` (default):

```
t.test(healthy, sick, var.equal=TRUE)
t.test(healthy, sick, var.equal=FALSE)
```

Paired t-test is used when values in both samples can be paired, e.g. one sample represents blood pressure of patients before and one sample after treatment. It is better to evaluate differences for individual patients one by one, rather than whole samples. The t-test can be switched to paired by `paired=TRUE`:

```
x<-rnorm(10, mean=20, sd=5)
x
y<-x+rnorm(10, mean=2, sd=0.5)
y
t.test(x,y)
t.test(x,y, paired=TRUE)
```

0.0.8.1 Tips and tricks

- the function `t.test` always gives a résumé on the alternative hypothesis, you can use it if you are not sure which variant of test should be used
- the function `t.test` (as well as `plot`) can use class 'formula' as the input. We will use it frequently in next lessons, so let us try it now:

```
healthy<-rnorm(10, mean=12.3, sd=3.3)
sick<-rnorm(10, mean=8.5, sd=3.3)
growth<-c(healthy, sick)
health<-rep(c("healthy", "sick"), each=10)
df<-data.frame(health, growth)
plot(growth~health, data=df)
t.test(growth~health, data=df)
```

1 Non-parametric tests

Up to now we considered normal distribution of a variable. Here we will show how we can test whether the variable follows the normal distribution and what can we do if data do not follow normal distribution. There is a graphical tool to do that known as QQ-plot.

```
qqnorm(x)
qqline(x)
```

The function calculates the mean and the standard deviation for the sample and from this it calculates quantiles. Then it plots the values of the sample vs. quantiles. This plot should be linear. If not, it means that the distribution is right or left skewed, bimodal or non-normal in some other way.

QQ-plot is good for visual evaluation, but for a quantitative evaluation it is useful to use some test of normality. One of them is the test developed by Shapiro and Wilk. You can run it by:

```
x<-rnorm(20)
shapiro.test(x)
```

Let us try with something non-normal

```
x<-c(rnorm(10), rnorm(10, mean=4))
shapiro.test(x)
```

The null hypothesis is that the sample follows the normal distribution.

What about if data do not follow the normal distribution? A non-parametric (i.e. not requiring normal distribution) alternative to t-test is Wilcoxon test. The two-sample variant is also known as Mann-Whitney test. We can use the function `wilcox.test`. It is used the same way as t-test:

```
x<-rnorm(10)
y<-rnorm(10, mean=2)
wilcox.test(x,y)
```

Let us try to compare with t-test:

```
t.test(x,y)
```

1.1 Tips and tricks

- non-parametric variant of analysis of variance will be shown later.

1.1.1 Analysis of Variance

Analysis of Variance (ANOVA) is an extension of t-test to more than two samples. The null hypothesis is that all samples have the same mean. We simply cannot do a pairwise t-test because this increases probability of rejection of the null hypothesis simply by chance. First, let us show a "manual" version of ANOVA for three samples, one representing a biological parameter of patients who were administered a drug, one representing a control group and one representing patients administered a placebo:

```
drug<-rnorm(10, mean=70, sd=30)
control<-rnorm(10, mean=100, sd=25)
placebo<-rnorm(10, mean=90, sd=25)
drug
control
placebo
```

First, let us calculate variances in each group:

```
sdrug<-sum((drug-mean(drug))^2)
scontrol<-sum((control-mean(control))^2)
splacebo<-sum((placebo-mean(placebo))^2)
```

We will sum this and we will call it sum of squares within the group (SSW):

```
SSW<-scontrol+sdrug+splacebo
```

Next, we will concatenate all samples and calculate mean of this supersample. We will call the variance of the supersample as sum of squares total (SST):

```
all<-c(control, drug, placebo)
SST<-sum((all-mean(all))^2)
```

In an extreme example that means of all samples are the same, the SSW and SST are the same, otherwise SST is bigger than SSW. The difference of SST and SSW is thus a measure of difference between the samples. We will cal this sum of squares between the groups (SSB):

```
SSB<-SST-SSW
```

The criterion with the F-distribution is calculated as:

```
FE<-(SSB*27)/(SSW*2)
FE
```

with two degrees of freedom, 27 is the total number of values minus number of samples (30-3), and 2 is number of samples minus 1. Finally, we will compare this with the value of the quantile of F-distribution:

```
qf(p=0.95, df1=2, df2=27)
```

If the `FE` is higher than `qf` we can reject the null hypothesis (i.e. that means of samples are the same).

In an automated way we can make use of a data frame:

```
labels<-rep(c("control", "drug", "placebo"), each=10)
all<-c(control, drug, placebo)
df<-data.frame(ident, all)
```

We will make a model by the function analysis of variance and we will obtain all results by summary of the model:

```
mymodel<-aov(all~labels, data=df)
mymodel
summary(mymodel)
```

We can reject the null hypothesis on the basis of the p-value. The same result can be obtained by:

```
anova(lm(all~labels, data=df))
```

Data for ANOVA must follow normal distribution and there must be homogeneous variances of samples. For other than normal distribution try data transformation or Kruskal-Walis test (bellow). For different variances try transformation.

1.2 Tips and Tricks

- to test normality of data you can use the function `mshapiro.test`. It is a multivariate alternative to Shapiro and Wilk test.
- to test homogeneity of variances you can use `bartlett.test` or `fligner.test`.
- Kruskal-Wallis rank sum test (`kruskal.test`) is a non-parametric alternative to ANOVA:

```
kruskal.test(all~labels, data=df)
```

- two-way ANOVA will be introduced together with linear models. # P-value adjustment and other approaches for multiple comparisons

In the previous chapter we have shown ANOVA on drug testing example:

```
drug<-rnorm(10, mean=70, sd=30)
control<-rnorm(10, mean=100, sd=25)
placebo<-rnorm(10, mean=90, sd=25)
labels<-rep(c("control", "drug", "placebo"), each=10)
all<-c(control, drug, placebo)
df<-data.frame(ident, all)
mymodel<-aov(all~labels, data=df)
mymodel
summary(mymodel)
```

This shows that there is a difference between the means. Next we want to know which samples are statistically significantly lower and higher. Again we cannot make a pairwise t-tests because of probability of rejection of the null hypothesis by chance. Instead we can use Tukey Honest Significance Test:

```
TukeyHSD(mymodel)
plot(TukeyHSD(mymodel))
```

Alternatively it is possible to do a pairwise t-test and adjust p-values by `pairwise.t.test`:

```
pairwise.t.test(all, labels, p.adjust.method="none")
```

The option `p.adjust.method` can be "none" (no adjustment), "bonferroni" (Bonferroni correction), "holm" (Holm and Bonferroni) and "BH" or "fdr" (Benjamini and Hochberg). There is also an option `pool.sd` defining whether variances are homogeneous.

1.3 Tips and Tricks

- in biological sciences we often compare every sample with a single control. For this it is useful to use Dunnett test. It requires package `multicomp`:

```
require(multcomp)
mydata <- data.frame(labels, all)
```

We must define that the first sample is the control:

```
mydata$labels <- relevel(mydata$labels, ref=1)
```

Finally, we will do the Dunnett test:

```
mydata.aov <- aov(all~labels, data=mydata)
mydata.dunnett <- glht(mydata.aov, linfct = mcp(labels="Dunnett"))
summary(mydata.dunnett)
```

Confidence intervals can be calculated as:

```
confint(mydata.dunnett)
```

You can also make a plot:

```
plot(mydata.dunnett)
```

- in biological sciences it is popular to use barplots and other similar plots with star symbols indicating the significance of a test, often with a horizontal lines that connect tested samples. In R you can generate such plots by the package "ggplot2" with "ggsignif" and "ggpubr".

2 Bi- and multivariate descriptive statistics

Let us generate a set of two well correlating variables `x` and `y`:

```
x<-1:10
y<-2:11+rnorm(10, sd=0.5)
x
y
plot(x,y)
```

Covariance can be calculated manually as:

```
sum((x-mean(x))*(y-mean(y)))/(length(x)-1)
```

Pearson correlation can be calculated as:

```
sum((x-mean(x))*(y-mean(y)))/sqrt(sum((x-mean(x))^2)*sum((y-mean(y))^2))
```

In R they can be calculated as:

```
cov(x,y)
cor(x,y)
```

Correlation can be calculated by dividing covariance by standard deviations of both variables:

```
cov(x,y)/(sd(x)*sd(y))
```

2.1 Tips and Tricks

- it is possible to apply these functions on data frame or matrix. This will make a pairwise correlation of all columns.

3 Linear models

In R you can use function `lm` to build a linear model. It can fit a dependent variable by one or multiple independent variables. Independent variables can quantitative, categorical or both.

```
x<-1:10
y<-2:11+rnorm(10, sd=0.5)
linfit <- lm(y~x)
linfit
```

This will show that "y" grows with "x". However, if you fit two noisy variables you will always obtain a result that "y" grows or descends with "x", there almost no chance to get zero slope even if the two variables completely uncorrelated. The question is whether the non-zero slope is statistically significant. This you can learn by function `summary`:

```
summary(linfit)
```

This is an extension of ANOVA in the way that the independent variable is not categorical (such as "control", "placebo" and "drug") but it is quantitative. The testing procedure is similar to that of ANOVA, the program calculates a sum of squares of error under assumption of null and alternative hypothesis and compares them.

There are several possibilities to describe models in the lm function:

function	expression in `lm`
f(x) = alpha	`y~1`
f(x) = alpha + beta x	`y~x`
f(x) = beta x	`y~-1+x`
f(x) = alpha + beta x + gamma xˆ2	`y~x+I(xˆ2)`
f(x) = alpha + beta_1 x_1 + beta_2 x_2	`y~x1+x2`
f(x) = alpha + beta_1 x_1 + beta_2 x_2 + gamma x_1 x_2	`y~x1*x2`

To get values of coefficients you can print coefficients:

```
linfit$coefficients
```

They can be iterated:

```
linfit$coefficients[1]
linfit$coefficients[2]
```

Alternatively you may use the function `coef`:

```
coef(linfit)[1]
coef(linfit)[2]
```

To plot a model into data you can use function `abline`:

```
plot(x,y)
abline(linfit)
```

The function `predict` predicts values of y for values of x based on the model. If you use:

```
newy<-predict(object=linfit)
```

it will calculate values of y for each x by a linear model. If you want to calculate this for some other values of x (here called "newx") you can type:

```
newx<-0:100/10
newy<-predict(object=linfit, newdata=data.frame(x=newx))
```

More interesting is calculation of confidence intervals. You can use:

```
newx<-0:100/10
newy<-predict(object=linfit, newdata=data.frame(x=newx), interval = 'prediction')
plot(x,y)
abline(linfit)
lines(newx, newy[,2], col="red")
lines(newx, newy[,3], col="red")
```

This will plot an interval to which 95 % of samples should fall (the level could be changed by option `level`). If you change "prediction" to "confidence" it will print confidence intervals for the model. Provided that there is some exact linear relationships between x and (inaccurately measured) y, we can accurately determine this relationship by doing an infinite number of measurements. If we do enough measurements we can get a vary narrow confidence interval:

```
newx<-0:100/10
newy<-predict(object=linfit, newdata=data.frame(x=newx), interval = 'confidence')
plot(x,y)
abline(linfit)
lines(newx, newy[,2], col="blue")
lines(newx, newy[,3], col="blue")
```

Prediction intervals are analogous to standard deviation, confidence intervals to standard error of the mean.

3.1 Tips and Tricks

- you can compare models "model1" and "model2" by `anova(model2, model1)`. This will tell you whether "model2" is significantly better than "model1".
- as already mentioned, linear models can use continuous as well as categorical independent variables. In order to do ANOVA with two factors use a linear model with `y~x1+x2`. In order to do ANOVA with two factors and their interactions use a linear model with `y~x1*x2`. Beside other ANOVA presumptions (normal distribution, homogeneity of variances) it is necessary to
- contingency tables are an alternative to ANOVA with categorical dependent and independent variables. To test statistical significance you can use `chisq.test` function.

4 Principal Component Analysis

Principal Component Analysis (PCA) is frequently used data analysis method. Let us demonstrate it on the results of Tour de France 2013. You can load the data by typing:

```
tdf <- read.table("https://raw.githubusercontent.com/spiwokv/Rtutorial/master/data/tourdefrance2013.
```

Every frame corresponds to a single rider (riders that did not finish the race were removed). The numbers in the table corresponds to their order in each stage. Each column corresponds to one stage. One stage was removed. It was a team time trial where order is given to teams rather than individual riders. It is very complicated to somehow visualize this data, because they are 20-dimensional (because of 20 stages). This is a great opportunity for PCA. Let us try on column 3-22 containing the results:

```
pcamodel <- prcomp(tdf[,3:22])
summary(pcamodel)
```

To plot importance of components you can use `plot` function on the model:

```
plot(pcamodel)
```

This shows that there is 1-2 important components describing the data very well. You can plot PC1 vs. PC2:

```
biplot(pcamodel)
```

Each number in black corresponds to the number of line in the `tdf` table. You can print their names as:

```
tdf[108,2]
```

to reveal the identity of the rider 108. The red arrows show the relationships between the original data and the results of the PCA. For example, some arrows point right and some point to the bottom. The former correspond to mountain stages. The later to flat stages.

From the results of PCA you can recognize that rider on left side of the plot are those who reached best in the overall classification of the race. These are typically slim mountain stage specialists who scored well in mountain stages (the arrow points right, more right means higher place in a mountain stage).

Riders at the top reached good results in flat stages. These are typically masculine riders who perform poorly in mountain stages. Riders in the bottom right cloud are "domestiques" who don't care much about their own results or rides who bet on just one stage and performed poorly in other stages.

4.1 Tips and Tricks

- the data can be centered and or normalized before PCA. Centering is a usual step in PCA and PCA without centering is rarely used. Normalization is used if you analyze apples and pears. They are controlled by: `center`, `scale.` (note: there is a dot after "scale").
- you can use PC1 and PC2 values using `predict` function, e.g.:

```
plot(predict(pcamodel))
```

To plot PC1 vs. PC3 use:

```
plot(predict(pcamodel)[,c(1,3)])
```

- in `biplot` you can control size of red arrows by `expand`. You can switch them off by setting it to zero:

```
biplot(pcamodel, expand=0)
```

5 Cluster Analysis

We can demonstrate cluster analysis on the same dataset as PCA:

```
tdf <- read.table("https://raw.githubusercontent.com/spiwokv/Rtutorial/master/data/tourdefrance2013.
```

First let us try non-hierarchical clustering by K-means method. For this we have to chose the number of cluster (the value of K). For example we have some feeling that there are 5 clusters of riders such as general classification specialists, sprinters, combined, domestiqeus and those who bet on just one or few stages. This can be done by:

```
kmodel <- kmeans(tdf[,3:22], 5)
kmodel
```

Each rider was placed into one of 5 clusters and the vector with this results can be printed as:

```
kmodel$cluster
```

We can illustrate the results on the plot from PCA:

```
pcamodel <- prcomp(tdf[,3:22])
plot(predict(pcamodel), col=rainbow(5)[kmodel$cluster], pch=20)
```

You can see that the distribution into clusters works very well.

Hierarchial clustering can be done by typing:

```
distances <- dist(tdf[,3:22])
hmodel <- hclust(distances)
plot(hmodel)
```

You can again inspect identities of riders to find which branch corresponds to sprinters, which corresponds to general classification specialists etc.

5.1 Tips and Tricks

- the algorithm used for K-means clustering can be controlled by `algorithm` option.
- the type of distance can be changes by `method` option of `dist` function.
- the parameter `method` controls the method used to construct the tree in `hclust`.

6 Programming

R is a programming language. You can make loops and other programming constructions in R.

6.1 Tips and Tricks

- you can make a for-loop by:

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

- always use `print` function to print something in a loop
- you can "if" statement by:

```
for(i in 1:10) {
  if(i==3) {
    print(i)
  }
}
```

If-else statement and switch statement works as well.

- you can define a function:

```
sem <- function(x) {
  return(sd(x)/sqrt(len(x)))
}
sem(rnorm(10))
```

- to run a program in R in a command line save the code to "program.R" and run it by typing:

```
Rscript program.R
```

or

```
R --no-save < program.R
```

- you can use a kind of object oriented programming in R
- you can use generic function, such as plot, defined to an object you define

7 Next Steps

7.1 Tips and Tricks

- you can use any or packages available in CRAN
- you can use tidyverse family of packages for data analysis
- you can use Bioconductor family of packages for data analysis
- you can read R Journal to learn more about news in R community and new R packages
- you can make your own package and submit it to CRAN.
- there are special R user conferences (UseR!, eRum, . . .)
- there are special R user communities such as R ladies