Introduction to R, Part 1

Basic preparation

This lesson contains basic information about R syntax and object types, data manipulation and structures and basic arithmetic and statistical operations

```
setwd("path/to/directory")
```

(in Linux, you can also simply start a terminal in whatever directory you want to work in, start R within the terminal, and you will be in that working directory automatically)

If in doubt as to whether you are already in the right directory, use:

```
getwd()
```

This will also tell you how the path should be formatted if you are unsure (e.g. windows' usage of backslashes\ as opposed to normal slashes/).

Operators, basic functions and objects

In R, you save data of various kinds internal variables referred to as "objects".

```
1->a #does the same as
a<-1
```

(arrow direction does not matter, use whatever you prefer)

You can use these objects like you would use numbers or variables in any normal mathematical operation, and save the output of any such operation to another variable. For example:

```
a<-1
b<-2
a+b->c
a*b->d
```

If you don't assign the results to any operation, default R behaviour is to print it to the screen (i.e. display it):

```
a #display content of a
#> [1] 1
```

```
a/b # display a/b
#> [1] 0.5
```

```
5/2 #normal division
#> [1] 2.5
```

```
5%/%2 #integer division
#> [1] 2
```

```r
trunc(5/2) #truncate a number, i.e. cut off decimal paces
#> [1] 2
```

```r
5%%2 #remainder for integer division
#> [1] 1
```

```r
b^b #b to the power of b
#> [1] 4
```

In addition to operators, we have already used a few functions (e.g. trunc() and setwd()). Functions in R are defined operations that use the workspace environment (all the objects stored in your current workspace) and/or the arguments (parameters) supplied within the parentheses:

```r
sqrt(4) #returns the square root of 4
#> [1] 2
```

```r
round(4.556776, 1) #rounds the number to the desired number of decimal paces
#> [1] 4.6
```

```r
signif(4.556776, 1) #rounds the number to the desired number of significant digits
#> [1] 5
```

```r

log(5) # natural log to base e
#> [1] 1.609438
```

```r
exp(log(5)) #this is e ^ log(5) = 5
#> [1] 5
```

```r
log10(5) #log to base 10, the inverse would be:
#> [1] 0.69897
```

```r
10^log10(5)
#> [1] 5
```

```r
#we can of course calculate logarithms to any arbitrary base using:
log10(100)/log10(3) #=log3(100)
#> [1] 4.191807
```

Functions can be very complex or very simple, for example sum() performs addition.

```r
c(1,2,3,4,5)->somenumbers
sum(somenumbers) #sum
#> [1] 15
```

```r
prod(somenumbers) #product
#> [1] 120
```

You can always recognize a function by it having parentheses(). Some functions also have a set of swirly brackets{} in addition to parentheses, in order to supply additional arguments, but this is rare and mostly used for conditional clauses or loops:

```r
a #what was a again?
#> [1] 1
```

```r

if(a==1){
print("a is 1")
}
#> [1] "a is 1"
```

We'll revisit if-clauses in more detail later. **NOTE**: if you enter a character-string (i.e. text), you **always** have to **use quotation marks**! if you had entered "print(a is 1)" you would have gotten an error:

```
print(a is 1)
#> Error: <text>:1:9: unexpected symbol
#> 1: print(a is
#>             ^
```

R uses the two letters NA (no quotation marks) to denote missing data:

```
NA->e
e
#> [1] NA
```

If you want to delete an object, use rm()

```
rm(d)
```

Every object has one or several classes of object. You can display them by using class():

```
class(c)
#> [1] "numeric"
```

Functions are simply objects of class "function"! For example:

```
class(rm)
#> [1] "function"
```

Besides that, the most basic classes are:

```
a<-"this is a text, class character"
class(a)
#> [1] "character"
```

```
a<-12345 # this is a number, class numeric
class(a)
#> [1] "numeric"
```

```
a<-TRUE # this is a boolean value (a logical True/False statement), class logical
class(a)
#> [1] "logical"
```

```
a<-as.Date("2025-01-24") #Date. Stored internally as a numeric value of days relative to an origin (def
class(a)
#> [1] "Date"
```

```
as.numeric(a)
#> [1] 20112
```

```
a<-as.POSIXct("2025-01-24 20:00:00", tz="CET") #POSIX-time, similar to a date but in seconds
class(a)
#> [1] "POSIXct" "POSIXt"
```

```
as.numeric(a)
#> [1] 1737745200
```

```
a<-factor(c("A","B","C")) # this is a categorical variable, class factor
class(a)
#> [1] "factor"
```

There are many other classes, some of which are part of packages, but they are mostly built upon, or contain, the previous few basic classes.

Complex objects

Objects can contain more than one single element. The simplest type of multi-element object is the vector. We can make one by combining the individual elements (or severel other vectors) using the c()-function:

```
c(1,b,c,2)->d
d #show our vector
#> [1] 1 2 3 2
```

We can also generate vectors using a sequence of numbers, or by repeating another object

```
c(1:10) #is a shorthand for:
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1,10,1)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

```
rep(5,5)
#> [1] 5 5 5 5 5
```

```
rev(d)#this reverses the order of the vector's elements
#> [1] 2 3 2 1
```

You determine the length of a vector as follows:

```
length(d) #this returns the number of elements, including NAs
#> [1] 4
```

```
sum(!is.na(c(1,2,3,NA)))#this returns only the number of elements that are not NA
#> [1] 3
```

Elements in a vector can be called up individually using the subset operator []:

```
d[1] #the first element of d
#> [1] 1
```

```
d[1:3] #the first three elements of d
#> [1] 1 2 3
```

Another type of data object is the matrix:

```
matrix(seq(1,12,1), nrow=3, ncol=4, byrow=T)->matrixA
#a matrix can also be made by binding multiple vectors together
cbind(c(1,2,3), c(4,5,6), c(7,8,9), c(10,11,12))->matrixB #binds several columns into a matrix
rbind(c(1,2,3,4), c(5,6,7,8), c(9,10,11,12))->matrixC #binds several rows into a matrix
```

Since the matrix is two-dimensional, when referring to elements we now need to specify both the row (first) and the column (second). If we want to select the entire row/column, we just leave the value empty

```
matrixA[1,] #select first row
#> [1] 1 2 3 4
```

```
matrixA[,1] #select first column
#> [1] 1 5 9
```

```
matrixA[1,1] #select cell in first row and first column
#> [1] 1
```

You can multiply compatible matrices using the operator for matrix multiplication:

```
matrixB %*% t(matrixC)
#>      [,1] [,2] [,3]
#> [1,]   70  158  246
#> [2,]   80  184  288
#> [3,]   90  210  330
```

Because we need the second matrix to have the same number of rows as the first matrix has columns, we need to use t() to transpose the matrix

An **array** is an extension of a matrix into a third dimension, i.e. a bunch of matrices of the same dimensionality stacked on top of each other:

```
array(c(matrixA, matrixB, matrixC), dim=c(3,4,3))
#> , , 1
#> 
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    5    6    7    8
#> [3,]    9   10   11   12
#> 
#> , , 2
#> 
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
#> 
#> , , 3
#> 
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    5    6    7    8
#> [3,]    9   10   11   12
```

A **data-frame** is similar to a matrix, but is more versatile and can contain several different types of data (e.g. numbers, text, logicals etc.). Technically speaking, a data.frame() is a special type of a list()-object that only contains vectors of the same length (i.e. different columns of a table)

```
data.frame(a=c(1,2,3), b=c(2,3,4), ID=c("first","second","third"))->dframe #here the ID-column contains
colnames(dframe) #this shows the column names of the data frame, and can also be used to modify them. F
#> [1] "a"  "b"  "ID"
```

```
colnames(dframe)<-c("a","b","ID")
```

The result should be a table like this:

a	b	ID
1	2	first
2	3	second
3	4	third

data.frames are just a special type of a class of object named a list. Lists can contain all sorts of other objects within them, including other lists/data.frames.

```
list()->l
l$a<-a
l$dframe<-dframe
```

Within lists, including data.frames, we can use the $-operator to find extract named sub-objects:

```
l$dframe
#>   a b     ID
#> 1 1 2  first
#> 2 2 3 second
#> 3 3 4  third
```

```
#this does the same as using the extraction operator: double square brackets [[]]
l[["dframe"]]
#>   a b     ID
#> 1 1 2  first
#> 2 2 3 second
#> 3 3 4  third
```

```
#we can also use single square brackets []
l["dframe"]
#> $dframe
#>   a b     ID
#> 1 1 2  first
#> 2 2 3 second
#> 3 3 4  third
```

#the difference between this and the subset operator isn't always obvious, but it helps to check the class of the returned object:

```
class(l["dframe"])#returns "list"
#> [1] "list"
```

```
class(l[["dframe"]])#returns "data.frame"
#> [1] "data.frame"
```

[[]] **extracts** an object from the parent object, while [] **subsets** the parent object. This becomes clearer when we apply it to columns within our data.frame:

```
class(dframe["ID"])#returns a single-column data.frame
#> [1] "data.frame"
```

```
class(dframe[["ID"]])#returns a character vector
#> [1] "character"
```

On the other hand, we can use $ and [[]] interchangeably, as long as the elements are named. But we need to make sure we use quotation marks with [[]]

```
l$dframe$ID #does the same as
#> [1] "first"  "second" "third"
```

```
l$dframe[["ID"]]
#> [1] "first"  "second" "third"
```

Return the number of rows or columns of any matrix or data.frame using:

```r
ncol(dframe)#n of columns
#> [1] 3
```

```r
nrow(dframe)#n of rows
#> [1] 3
```

Now that we have created a bunch of useless objects, you might want to list them all:

```r
ls()
#>  [1] "a"                     "A"                     "add.alpha"
#>  [4] "ages_sptab"            "akaike.w"              "arbl"
#>  [7] "aspect_ratio"          "axial_total"           "b"
#> [10] "B"                     "bbmean"                "bootCI"
#> [13] "c"                     "cc"                    "ci.lm"
#> [16] "col_asign"             "col1"                  "coord"
#> [19] "correction_factor"     "cplr"                  "cptd"
#> [22] "cQE_mod"               "d"                     "datamatch"
#> [25] "DD"                    "dext"                  "dext_"
#> [28] "dframe"                "dfxy"                  "distr.sample"
#> [31] "e"                     "ebar"                  "ellipse"
#> [34] "ellipse.c"             "even"                  "export"
#> [37] "fdir"                  "forelimb"              "forelimb_lateral"
#> [40] "fun_tmp"               "geomean"               "ggcol"
#> [43] "highlight"             "hindlimb"              "hindlimb_lateral"
#> [46] "i"                     "ifs"                   "import.tree.lab"
#> [49] "ini"                   "install.if.missing"    "intercol"
#> [52] "jackCI"                "l"                     "lab.lm"
#> [55] "laglead"               "lmxy"                  "matrixA"
#> [58] "matrixB"               "matrixC"               "matrixplot"
#> [61] "measurements_dorsal"   "measurements_lateral"  "MM"
#> [64] "modelp"                "odd"                   "p_bootCI"
#> [67] "pAgreaterB"            "pie_"                  "plateo"
#> [70] "plotbg"                "plotr"                 "pred.lmxy"
#> [73] "regres"                "rowmeans"              "scalecent"
#> [76] "se"                    "se.prop"               "se2.3"
#> [79] "se3"                   "sil"                   "somenumbers"
#> [82] "tally_div"             "ua_substr"             "varCI"
#> [85] "varsum"                "vector_abc"            "x"
#> [88] "y"                     "YY"
```

We won't have time to explain every single function and parameter, but you can always call up the help page to every (documented) function by simply prepending a ?:

```r
?cbind()
?cbind
```

Reading and writing files

Directly save and load R objects to file using:

```r
save(l, file="l.RData") # save the object l to "l.rdata"
load("l.RData") # load the object l from "l.rdata"
save.image() # save entire workspace, defaults to ".RData"
```

Export R objects to text files/tables:

```
write(c,"c.txt")
write.table(dframe, "dframe.txt")
```

```
sum(c(1,2,3))->a
print(a)
#> [1] 6
```

```
##write the example to script file:
write("sum(c(1,2,3))->a\nprint(a)", file="examplescript.R")# \n inserts a line break. You could of cour

source("examplescript.R")#load the script we just saved
#> [1] 6
```

Basic statistical operations and plots

Most common statistical operations have their own functions, usually with easy to remember names, e.g.:

```
mean(c(1,2,3))# calculate the arithmetic mean
#> [1] 2
```

```
mean(c(1,2,3,NA), na.rm=T)# add na.rm=T if you have NA values that you want to ignore
#> [1] 2
```

```
sd(c(1,2,3,NA), na.rm=T) #same for standard deviation
#> [1] 1
```

```
var(c(1,2,3,NA), na.rm=T) #or variance
#> [1] 1
```

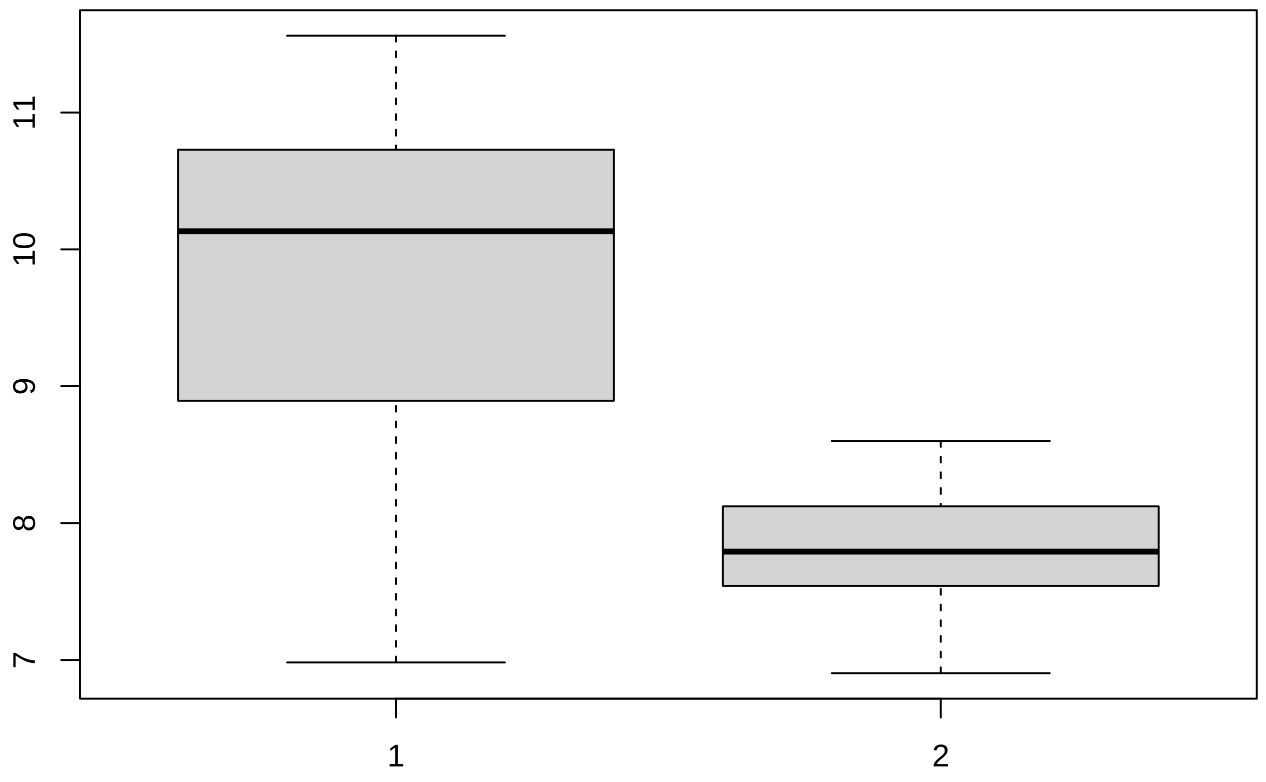
…you get the ghist.

Now we'll generate some random data to run a few examples on:

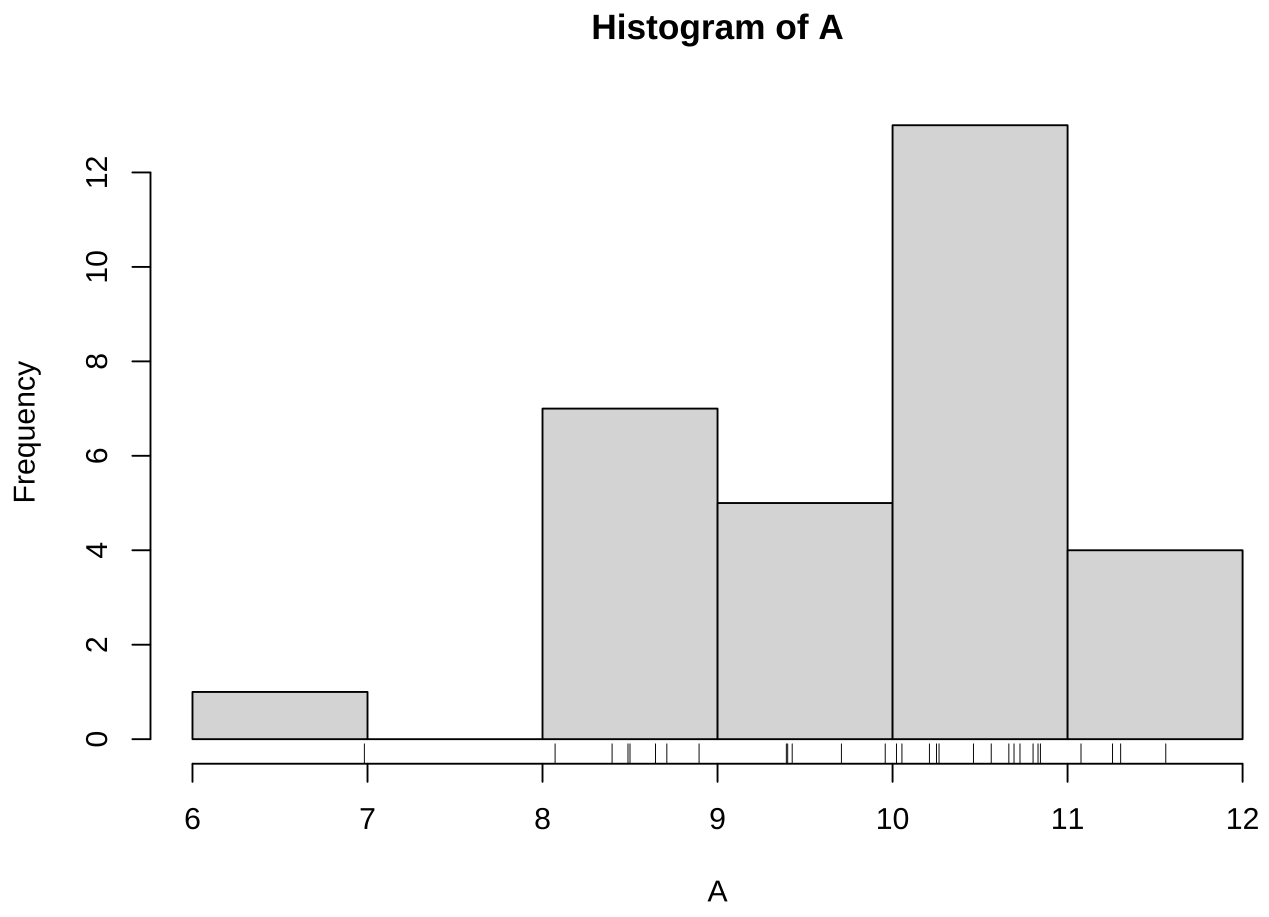
```
A<-rnorm(n=30,mean=10,sd=1)#rnorm samples the desired number of points from a normal distribution. Resu
B<-rnorm(n=30,mean=8, sd=0.5)
```

Do A and B differ significantly? First, let's look at some figures:

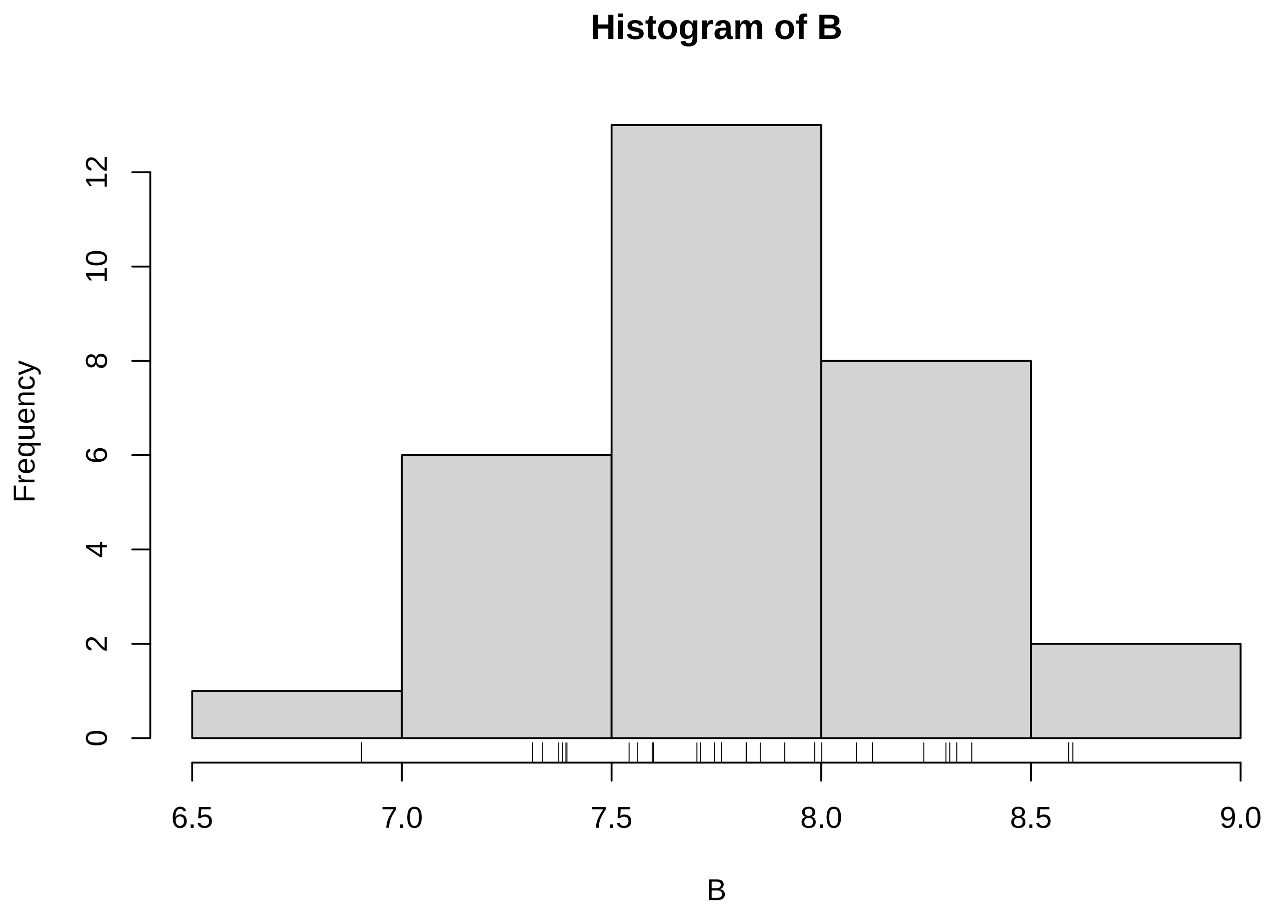
```
boxplot(A,B)
```



```
hist(A)
rug(A)
```



```r
hist(B)
rug(B)
```



If we want to test this numerically, R offers a variety of tools for this. Since these are normally distributed values, we can use the most basic one, a t-test:

```
t.test(A,B, conf.level=0.95, var.equal=F)#welch t-test, unless var.equal=T (then students t test)
#>
#>  Welch Two Sample t-test
#>
#> data:  A and B
#> t = 9.4042, df = 36.798, p-value = 2.515e-11
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>  1.608976 2.492919
#> sample estimates:
#> mean of x mean of y
#>  9.872294  7.821347
```

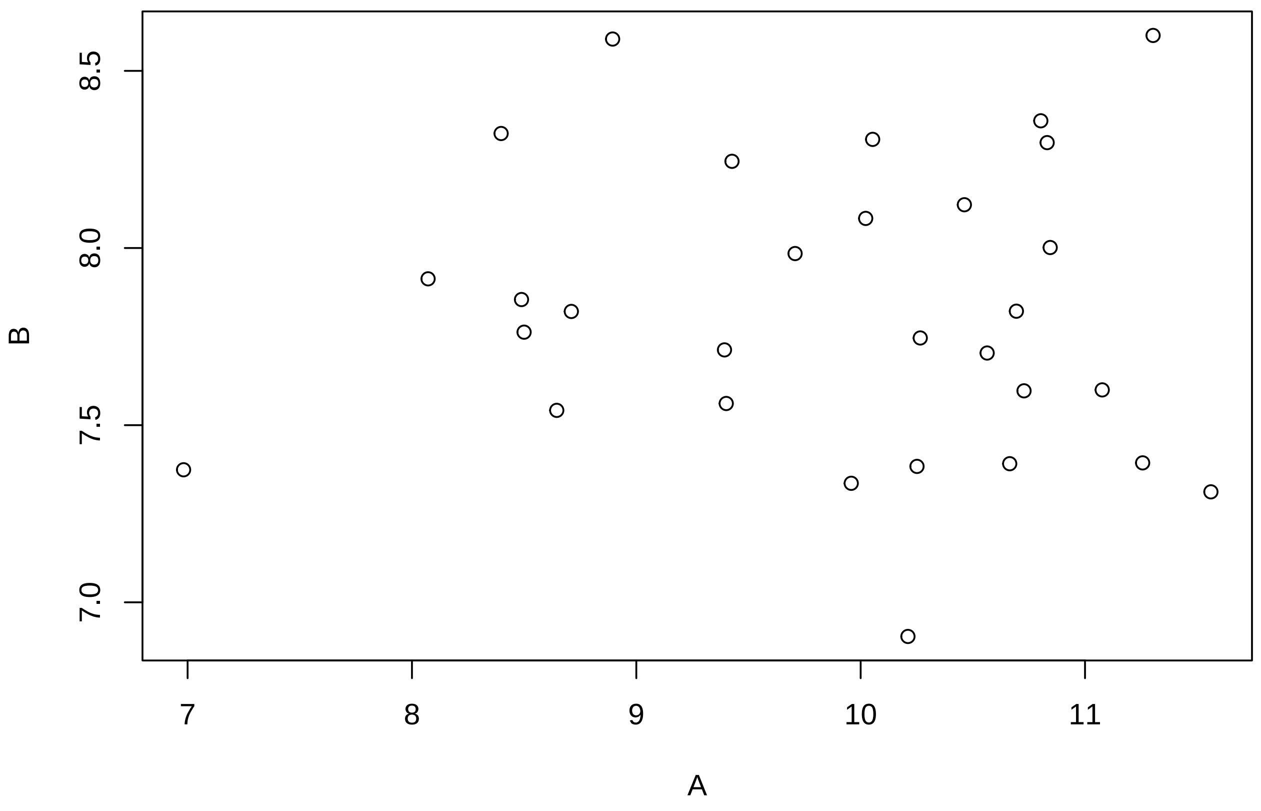
```
var.test(A, B, conf.level=0.9)#test for inequality of variances (if equal, do students t test)
#>
#>  F test to compare two variances
#>
#> data:  A and B
#> F = 7.3013, num df = 29, denom df = 29, p-value = 7.428e-07
#> alternative hypothesis: true ratio of variances is not equal to 1
#> 90 percent confidence interval:
#>   3.923708 13.586308
#> sample estimates:
```

```
#> ratio of variances
#>           7.301282
```

```
wilcox.test(A,B) # The wilcoxon-rank-sum-test would be appropriate to use on non-normally distributed d
#>
#>  Wilcoxon rank sum exact test
#>
#> data:  A and B
#> W = 856, p-value = 7.593e-12
#> alternative hypothesis: true location shift is not equal to 0
```

We can also pretend our data aren't two univariate samples but rather one bivariate one and visualize them as a scatterplot:

```
plot(A,B, type="p")
```



Topics covered

Understand the basic types of objects: functions (function), numbers (numeric), text (character), and how to tell R which is which

Understand types of data objects: vectors (1D), matrices and data.frames (2D) and arrays (3D) as well as lists() and how they can be used to save and structure your data

Be able to use R like a normal calculator to perform basic operations +-*/^

Be able to build vectors, matrices or data.frames with your own data, read text files with data and save them in appropriate data objects or files

Apply basic functions and calculate basic statistical parameters on the data, e.g. log(), mean(), sd(), t.test()…

Be able to make simple plots such as boxplots and histograms