Statistical programming using R

Lecture 2 Data loading, plotting and analysis

Reading and Writing Data

Reading data

scan()

Function `scan()` is the most simple and straight-forward function to load data from the file or console. Data will be stored in a vector or a list of vectors, so it is important to control for the same type of data in each vector. Press "Enter" twice to indicate an end of data entry.

```
> numbers <- scan()
1: 1 2 3 4
5: 5 6
7:
Read 6 items
> numbers
[1] 1 2 3 4 5 6
```

This is a default behavior for `scan()` to take numerical values from the terminal. Or saying that more precisely, to take values from the terminal and convert them into numbers. You can take character values too and there is not need to use quotation marks in the input.

```
> pets <- scan(what="")
1: cat dog bird
4: fish
5:
Read 4 items
> pets
[1] "cat"  "dog"  "bird" "fish"
```

You can load different data types to be stored in different vectors. In this case, the result is a list of vectors.

```
> pets <- scan(what = list("", 0, 0))
1: dog 1 2
2: cat 3 4
3: fish 5 6
4:
Read 3 records
> pets
[[1]]
[1] "dog"  "cat"  "fish"
```

```
[[2]]
[1] 1 3 5

[[3]]
[1] 2 4 6
```

It is possible to use `scan()` for loading data from the file.

```
# prepare data file for the example
cat("TITLE extra line",
    "2 3 5 7",
    "11 13 17", file = "ex.data.txt", sep = "\n")

# read the text file as numeric (default settings)
my.data <- scan("ex.data.txt", skip = 1, quiet = TRUE)
print(my.data)
```

```
## [1]  2  3  5  7 11 13 17
```

```
# read the text file as character with three columns
# with space as a delimiter (default settings)
my.data <- scan("ex.data.txt", what = list("","",""), skip = 1, nlines=2)
```

You get a warning as the data can not be properly split in three columns. Warning is not an error. Your code execution would not stop after the warning, unlike an error that stops the program. Warning is an indication that something might be not right and you should double check everything. If you are confident that everything is OK, then you can ignore the warning or you can even suppress the warning.

```
print(my.data)
```

```
## [[1]]
## [1] "2"  "7"  "17"
##
## [[2]]
## [1] "3"  "11" ""
##
## [[3]]
## [1] "5"  "13" ""
```

```
unlink("ex.data.txt") # tidy up - close connection
```

Function `scan()` is a universal function and it can be used to read any text files. There are other functions that do a similar job, for example `readLines()` to read from the file or `readline()` to read from the terminal. However, you will not use these functions too often as there are many specialised functions that can do reading data from files much better.

`read.table()`

Examples in the previous section were about (potentially) unstructured data. Very often text files with data are semi-structured data with a fixed number of row and columns. Function `read.table()` is one of the most important for loading data into R. It reads a text file and converts loaded data into a data frame.

```r
# read data from the file, use comma as delimiter
# first row has headings and first column has names of rows
my.data <- read.table(file="mtcars.csv", header=TRUE, sep=",", row.names=1)
head(my.data)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

As you can see, data was loaded into a data frame but also all columns were converted into appropriate data types - doubles and integers.

In most cases this advanced functionality works fine. However, sometimes automatic conversion is not desirable. In particular, when loading character (string) type of variables. By default, the function will try to convert any character column in to a `factor`. To stop that you can adjust a parameter `stringsAsFactors` and make it `FALSE`.

Please check a help file for `read.table()` - there are a huge number of parameters. To make a life a bit easier there are several wrapper functions around `read.table()` that can be used in somewhat easy or common situations when you don't need fine-tuning function parameters. For example:

```r
# read data from the csv-file
my.data <- read.csv(file="mtcars.csv")
head(my.data)
```

```
##               model  mpg cyl disp  hp drat    wt  qsec vs am gear carb
## 1         Mazda RX4 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## 2     Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## 3        Datsun 710 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## 4    Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## 5 Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## 6           Valiant 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Other functions from the same family are `read.csv2()`, `read.delim()`, `read.delim2()`. All of them are based on `read.table()` with most parameters predefined.

Loading other data formats

Data presented to you can be stored in a huge number of different formats and most probably you will be able to find a package that can open that format and load data into R. Here is just a brief list.

Text files Besides presented above functions from the `base` package, there are other functions that can do the same job of loading text files and do it better!

```r
library(readr)

read_csv()    # for comma-separated values (CSV) files
read_tsv()    # for tabilation delimited files
read_delim()  # for any type of delimiters
```

Excel files There are several packages to load Excel files that do the job with more or less success: `xslx, readxl, XLConnect, openxlsx`. Every package has function `read.xlsx()` or `read_excel()` or something similar to allow you loading data from the Excel file including Excel files with multiple working sheets.

Statistical packages R can read files in SAS (`sas7bdat`). Also, there is a couple universal packages (`foreign, haven`) that can read different formats at the same time – SAS, SPSS, State, Weka, dBase, Mintab, etc.

Writing data

Most of the packages listed above can read and write(!!!) data in these "foreign" formats. Hence, you can create files natively supported by other applications. Look for the functions like (beware of a proper package for each function):

```
write.csv()               # CSV format, it is universally supported by all applications
readr::write_csv()        # CSV format again
openxlsx::write.xlsx()    # Excel
foreign::write.arff()     # Weka

# for other statistical packages
foreign::write.foreign(df, datafile, codefile, package = c("SPSS", "Stata", "SAS"), ...)
```

Read and write data in R-format

R has its own format to store data. And "data" mean multiple variables at the same time.

```
x <- 2
y <- list(a=1:3, b=LETTERS[1:3])

save(x,y, file="mydata.RData")   # save two variables
rm(list=ls())                    # delete all variables - clean up everything
load(file="mydata.RData")        # load stored data
print(y)                         # check results
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] "A" "B" "C"
```

You can even store your entire working environment with all loaded variables and functions.

```
myfun <- function(x){
  print(x)
}
save.image(file="myimage.RData") # save the working environment
rm(list=ls())                    # delete all variables - clean up everything
load(file="myimage.RData")       # restore the working environment
myfun(x)                         # check results
```

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

Working directory

When saving or loading any files, R always look in the working directory unless you specify an absolute path to the file or folder. The last one is a very bad practice. The good approach is to put your files in the working directory and use a relative path.

```
getwd()                 # to check your current working directory
setwd("C:/Users/tim")   # to set working directory
```

Even better way to deal with working directory is to use RStudio functionality. Go into a menu `Session -> Set Working Directory -> To Source File Location`. This way your working directory will be the same one where your R-code file is stored.

Warnings Suppression

Warnings in R might be a very useful tool to better understand the running process or the function, to improve your code or even to avoid costly mistakes.

```r
x <- c("1", "2", "a")
y <- as.numeric(x) # this results in a warning as you try to convert "a" into a number
```

At the same time, warnings might be annoying and useless if you are 110% confident that your code works as it should. In this case, you can suppress warnings. There are two ways to do that.

1. You can change global settings and suppress all warnings from all functions in your code.

```r
options(warn = -1)
y <- as.numeric(x)      # no warning from this function and
print(y)                # from any other function in the code
```

```
## [1]  1  2 NA
```

You can run function `options()` without any parameters to see your current global settings and then you can change any of them. For example, you can change `stringsAsFactors` to `FALSE` and avoid hassle with automatic conversion of strings into factors in all data frames mentioned in the previous lecture, and in loading text files discussed above.

2. You can change settings only for one function you run and keep warnings allowed for all other functions in the code

```r
options(warn = 0)       # restore warnings on global level
y <- as.numeric(x)      # this results in a warning as before

y <- suppressWarnings(as.numeric(x))  # evaluate the function but ignore all warnings
print(y)
```

```
## [1]  1  2 NA
```

The *apply family

R functions work well processing large amounts of data. For example,

```
x <- rnorm(1000)    # generate 1000 random numbers
print(mean(x))      # get an average of these numbers
```

```
## [1] -0.03266755
```

If you have a complex data structure with several data sets you might want to use the same type of analysis for each data set. It is possible to use a `for` loop:

```
my.data <- data.frame(x=rnorm(1000), y=rnorm(100))  # generate two sets of random numbers
for(i in my.data){
  print(mean(i))
}
```

```
## [1] 0.03169723
## [1] 0.08887525
```

This is a working solution but you should always try to avoid it. Using `for` loop is very inefficient. There is a family of `apply` functions that allow you to apply any R function (including custom functions) to multiple elements of any complex data structure.

The very first one is `apply()`. Its syntax as following:

```
apply(X, MARGIN, FUN, ...)
```

where `X` is an array-type object, e.g. data frame or matrix; `MARGIN` indicates a dimension for applying a function, 1 indicates rows, 2 indicates columns; `FUN` is a function to apply; `...` optional arguments for the function `FUN`.

```
# the same data as above, the same results
# however different format - a vector
apply(my.data, 2, mean)
```

```
##          x          y
## 0.03169723 0.08887525
```

Now let's try to use data set `mtcars` as before.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
apply(mtcars, 2, mean)  # calculate a mean for every column
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec 
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750 
##         vs         am       gear       carb 
##   0.437500   0.406250   3.687500   2.812500
```

The result is a named vector of means for each column. It is possible to calculate a mean for rows but it does not make any sense for the analysis.

```r
apply(mtcars, 1, mean)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive 
##            29.90727            29.98136            23.59818            38.73955 
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D 
##            53.66455            35.04909            59.72000            24.63455 
##            Merc 230            Merc 280           Merc 280C          Merc 450SE 
##            27.23364            31.86000            31.78727            46.43091 
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental 
##            46.50000            46.35000            66.23273            66.05855 
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla 
##            65.97227            19.44091            17.74227            18.81409 
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28 
##            24.88864            47.24091            46.00773            58.75273 
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa 
##            57.37955            18.92864            24.77909            24.88027 
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E 
##            60.97182            34.50818            63.15545            26.26273
```

The result is a vector of the same length as a number of rows.

You can use custom functions and even anonymous functions, for example to calculate a range for every column and then round it to one decimal place:

```r
apply(mtcars, 2, function(x) round(max(x)-min(x), 1))
```

```
##   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb 
##  23.5   4.0 400.9 283.0   2.2   3.9   8.4   1.0   1.0   2.0   7.0
```

If the function does not aggregate the data as it is done by `mean` or `sum` or `sd`, then the result has the same shape and format as the original data. Also, if a function requires extra arguments you can provide them.

```r
log.cars <- apply(mtcars, 2, log, base=3) # apply log trasformation with base 3 to every column
head(log.cars, 3)
```

```
##                    mpg     cyl     disp       hp     drat        wt     qsec
## Mazda RX4     2.771244 1.63093 4.619622 4.278562 1.238814 0.8767190 2.549519
## Mazda RX4 Wag 2.771244 1.63093 4.619622 4.278562 1.238814 0.9612606 2.579972
## Datsun 710    2.846100 1.26186 4.261860 4.125750 1.227069 0.7660275 2.661266
##                 vs am    gear    carb
## Mazda RX4     -Inf  0 1.26186 1.26186
## Mazda RX4 Wag -Inf  0 1.26186 1.26186
## Datsun 710       0  0 1.26186 0.00000
```

Next members of the apply family are `sapply()` and `lapply()`. Both of them consider elements of the vector or a list (list is a "special" vector too) and apply a function to every element. Hence, these functions do not require `MARGIN` parameter as vectors and lists have no dimensionality. The difference between these functions is a resulted data structure. It is a list for `lapply` and a vector for `sapply`. Here are some examples:

```
my.data <- list(a=rnorm(100), b=letters)
sapply(my.data, length)   # check the length of every element of my.data, result is a vector
```

```
##   a   b
## 100  26
```

```
lapply(my.data, length)   # the same as above but result is a list
```

```
## $a
## [1] 100
##
## $b
## [1] 26
```

You can use `saaply` and `lapply` with a data frame too. This is due to a data frame being a special list where every element is a column.

```
sapply(mtcars, mean)   # result is the same as above with function apply()
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

Obviously, you can not use `sapply` and `lapply` to do anything with rows of the data frame.

Above three functions are the most popular. However, there are some other members of the family.

There is a function `vapply` which is the almost same as `sapply` but it has an extra parameter `FUN.VALUE` to specify types of returned values. As a result of the extra information, this function can be safer and sometimes faster to run.

```
vapply(my.data, FUN=length, FUN.VALUE=0L)   # returned value should be an integer
```

```
##   a   b
## 100  26
```

Following members are even more exotic and not so common.

Function `tapply` break a variable into groups based on the other variable and then apply a given function to every group.

```
x <- 1:20    # define a vector
print(x)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```r
y <- factor(rep(letters[1:4], each = 5)) # define another vector of the same length
print(y)                                 # there looks to be 4 groups
```

```
##  [1] a a a a a b b b b b c c c c c d d d d d
## Levels: a b c d
```

```r
tapply(x, y, sum)   # get sum of elements of "x" corresponding to groups set by "y"
```

```
##  a  b  c  d
## 15 40 65 90
```

Function `tapply` can be used for data frames columns too. For example, you can use `mtcars` data to calculate an average fuel consumption in miles per gallon (US style) for cars with different number of cylinders.

```r
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

As you can see, 4 cylinder cars are the most economical as they make more that 26 miles per gallon of fuel.

While this functionality is extremely useful, `tapply` is not very popular function as this job is a very primitive example of groupby and aggregate functionality and there are functions that can be done this job much-much better. You will see these functions later.

Function `mapply` allows to work with multiple variables at the same time and apply a given function to the first element of the every variable, then to the second element of the every variable, to the third . . . , and so on. Results will be combined in one vector as in `sapply` function.

```r
# prepare data for the example
x <- 1:10; y <- 11:20; z <- 21:30
mapply(sum, x, y, z)   # do summation for elements of each variable
```

```
##  [1] 33 36 39 42 45 48 51 54 57 60
```

Function `rapply` works with elements of the nested list recursively and you can select format of the result object - a list or a vector.

```r
# prepare a nested list - there are three levels of lists
my.data <- list(list(a = 1:10, b = list(c = letters[1:5])), d = "a test")
rapply(my.data, length, how="replace")  # replace elements of the original list by the results
```

```
## [[1]]
## [[1]]$a
## [1] 10
##
## [[1]]$b
## [[1]]$b$c
## [1] 5
##
##
##
## $d
## [1] 1
```

```r
rapply(my.data, length, how="unlist")   # put results in a vector
```

```
##   a b.c   d
##  10   5   1
```

Statistical analysis - descriptive statistics

R is statistical programming language. Statistics is where R really shines. Let's start with descriptive statistics. Obviously, there are functions to get any statistics you might need. These functions can be used for an individual data set (just a column) or applied to multiple columns. You have already seen some examples. Many functions are included in a `base` package. You can get much-much-much more by using other packages.

```r
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```r
mean(mtcars$mpg)  # mean
```

```
## [1] 20.09062
```

```r
sd(mtcars$mpg)    # standard deviation
```

```
## [1] 6.026948
```

```r
var(mtcars$mpg)   # variance
```

```
## [1] 36.3241
```

```r
library(moments)       # load a library
skewness(mtcars$mpg)   # skewness
```

```
## [1] 0.6404399
```

```r
kurtosis(mtcars$mpg)   # kurtosis
```

```
## [1] 2.799467
```

```r
# you can get moments of any order
moment(mtcars$mpg, order=2, central = TRUE)
```

```
## [1] 35.18897
```

You can get all main statistics for all columns in your data set at the same time.

```r
fBasics::basicStats(mtcars)    # beware of the package required
```

```
##                    mpg        cyl         disp          hp       drat
## nobs         32.000000  32.000000    32.000000   32.000000  32.000000
## NAs           0.000000   0.000000     0.000000    0.000000   0.000000
## Minimum      10.400000   4.000000    71.100000   52.000000   2.760000
## Maximum      33.900000   8.000000   472.000000  335.000000   4.930000
## 1. Quartile  15.425000   4.000000   120.825000   96.500000   3.080000
## 3. Quartile  22.800000   8.000000   326.000000  180.000000   3.920000
## Mean         20.090625   6.187500   230.721875  146.687500   3.596562
## Median       19.200000   6.000000   196.300000  123.000000   3.695000
## Sum         642.900000 198.000000  7383.100000 4694.000000 115.090000
## SE Mean       1.065424   0.315709    21.909473   12.120317   0.094519
## LCL Mean     17.917679   5.543607   186.037211  121.967950   3.403790
## UCL Mean     22.263571   6.831393   275.406539  171.407050   3.789335
## Variance     36.324103   3.189516 15360.799829 4700.866935   0.285881
## Stdev         6.026948   1.785922   123.938694   68.562868   0.534679
## Skewness      0.610655  -0.174612     0.381657    0.726024   0.265904
## Kurtosis     -0.372766  -1.762120    -1.207212   -0.135551  -0.714701
##                     wt       qsec        vs        am       gear      carb
## nobs         32.000000  32.000000 32.000000 32.000000  32.000000 32.000000
## NAs           0.000000   0.000000  0.000000  0.000000   0.000000  0.000000
## Minimum       1.513000  14.500000  0.000000  0.000000   3.000000  1.000000
## Maximum       5.424000  22.900000  1.000000  1.000000   5.000000  8.000000
## 1. Quartile   2.581250  16.892500  0.000000  0.000000   3.000000  2.000000
## 3. Quartile   3.610000  18.900000  1.000000  1.000000   4.000000  4.000000
## Mean          3.217250  17.848750  0.437500  0.406250   3.687500  2.812500
## Median        3.325000  17.710000  0.000000  0.000000   4.000000  2.000000
## Sum         102.952000 571.160000 14.000000 13.000000 118.000000 90.000000
## SE Mean       0.172968   0.315890  0.089098  0.088210   0.130427  0.285530
## LCL Mean      2.864478  17.204488  0.255783  0.226345   3.421493  2.230158
## UCL Mean      3.570022  18.493012  0.619217  0.586155   3.953507  3.394842
## Variance      0.957379   3.193166  0.254032  0.248992   0.544355  2.608871
## Stdev         0.978457   1.786943  0.504016  0.498991   0.737804  1.615200
## Skewness      0.423146   0.369045  0.240258  0.364016   0.528854  1.050874
## Kurtosis     -0.022711   0.335114 -2.001938 -1.924741  -1.069751  1.257043
```

```r
# another and more detailed version
Hmisc::describe(mtcars)        # beware of the package required
```

```
## mtcars
##
##  11  Variables      32  Observations
## --------------------------------------------------------------------------------
## mpg
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       25    0.999    20.09    6.796    12.00    14.34
##      .25      .50      .75      .90      .95
##    15.43    19.20    22.80    30.09    31.30
```

```
##
## lowest : 10.4 13.3 14.3 14.7 15.0, highest: 26.0 27.3 30.4 32.4 33.9
## ---------------------------------------------------------------------------
## cyl
##        n  missing distinct     Info     Mean      Gmd
##       32        0        3    0.866    6.188    1.948
##
## Value          4     6     8
## Frequency     11     7    14
## Proportion 0.344 0.219 0.438
## ---------------------------------------------------------------------------
## disp
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       27    0.999    230.7    142.5    77.35    80.61
##      .25      .50      .75      .90      .95
##   120.83   196.30   326.00   396.00   449.00
##
## lowest :  71.1  75.7  78.7  79.0  95.1, highest: 360.0 400.0 440.0 460.0 472.0
## ---------------------------------------------------------------------------
## hp
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       22    0.997    146.7    77.04    63.65    66.00
##      .25      .50      .75      .90      .95
##    96.50   123.00   180.00   243.50   253.55
##
## lowest :  52  62  65  66  91, highest: 215 230 245 264 335
## ---------------------------------------------------------------------------
## drat
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       22    0.997    3.597   0.6099    2.853    3.007
##      .25      .50      .75      .90      .95
##    3.080    3.695    3.920    4.209    4.314
##
## lowest : 2.76 2.93 3.00 3.07 3.08, highest: 4.08 4.11 4.22 4.43 4.93
## ---------------------------------------------------------------------------
## wt
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       29    0.999    3.217    1.089    1.736    1.956
##      .25      .50      .75      .90      .95
##    2.581    3.325    3.610    4.048    5.293
##
## lowest : 1.513 1.615 1.835 1.935 2.140, highest: 3.845 4.070 5.250 5.345 5.424
## ---------------------------------------------------------------------------
## qsec
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##       32        0       30        1    17.85    2.009    15.05    15.53
##      .25      .50      .75      .90      .95
##    16.89    17.71    18.90    19.99    20.10
##
## lowest : 14.50 14.60 15.41 15.50 15.84, highest: 19.90 20.00 20.01 20.22 22.90
## ---------------------------------------------------------------------------
## vs
##        n  missing distinct     Info      Sum     Mean      Gmd
##       32        0        2    0.739       14   0.4375   0.5081
```

```
##
## --------------------------------------------------------------------------------
## am
##        n  missing distinct     Info      Sum     Mean      Gmd
##       32        0        2    0.724       13   0.4062    0.498
##
## --------------------------------------------------------------------------------
## gear
##        n  missing distinct     Info     Mean      Gmd
##       32        0        3    0.841    3.688   0.7863
##
## Value          3     4     5
## Frequency     15    12     5
## Proportion 0.469 0.375 0.156
## --------------------------------------------------------------------------------
## carb
##        n  missing distinct     Info     Mean      Gmd
##       32        0        6    0.929    2.812    1.718
##
## lowest : 1 2 3 4 6, highest: 2 3 4 6 8
##
## Value          1     2     3     4     6     8
## Frequency      7    10     3    10     1     1
## Proportion 0.219 0.312 0.094 0.312 0.031 0.031
## --------------------------------------------------------------------------------
```

```r
# one more version
psych::describe(mtcars)       # beware of the package required
```

```
##      vars  n   mean     sd median trimmed    mad   min    max  range  skew
## mpg     1 32  20.09   6.03  19.20   19.70   5.41 10.40  33.90  23.50  0.61
## cyl     2 32   6.19   1.79   6.00    6.23   2.97  4.00   8.00   4.00 -0.17
## disp    3 32 230.72 123.94 196.30  222.52 140.48 71.10 472.00 400.90  0.38
## hp      4 32 146.69  68.56 123.00  141.19  77.10 52.00 335.00 283.00  0.73
## drat    5 32   3.60   0.53   3.70    3.58   0.70  2.76   4.93   2.17  0.27
## wt      6 32   3.22   0.98   3.33    3.15   0.77  1.51   5.42   3.91  0.42
## qsec    7 32  17.85   1.79  17.71   17.83   1.42 14.50  22.90   8.40  0.37
## vs      8 32   0.44   0.50   0.00    0.42   0.00  0.00   1.00   1.00  0.24
## am      9 32   0.41   0.50   0.00    0.38   0.00  0.00   1.00   1.00  0.36
## gear   10 32   3.69   0.74   4.00    3.62   1.48  3.00   5.00   2.00  0.53
## carb   11 32   2.81   1.62   2.00    2.65   1.48  1.00   8.00   7.00  1.05
##      kurtosis    se
## mpg     -0.37  1.07
## cyl     -1.76  0.32
## disp    -1.21 21.91
## hp      -0.14 12.12
## drat    -0.71  0.09
## wt      -0.02  0.17
## qsec     0.34  0.32
## vs      -2.00  0.09
## am      -1.92  0.09
## gear    -1.07  0.13
## carb     1.26  0.29
```

These are just some examples. There might be thousands more. No need to memorise all of them. Just pick whatever works best for you. Don't load too many packages at the same time as there might be a conflict between different functions with the same name as in the example above.

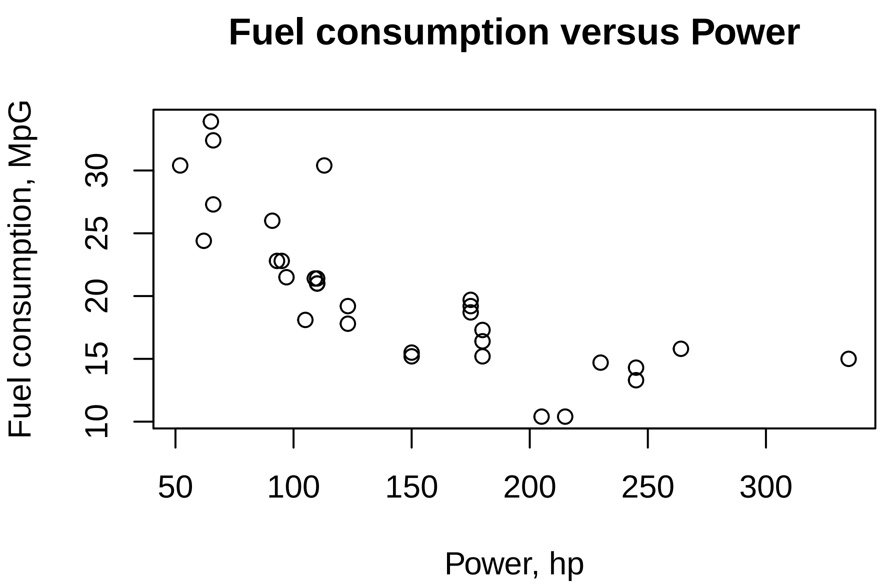
Basic plotting

Plotting capability is another major advantage of R compare to other programming languages. Plotting is relatively easy and even a basic plotting is quite advanced. However, later you will see even better and more advanced plotting.

Scatter plot

Scatter plot is used to study a relationship between two numerical variables

```
plot(x=mtcars$hp, y=mtcars$mpg, main="Fuel consumption versus Power",
     xlab="Power, hp", ylab="Fuel consumption, MpG")
```

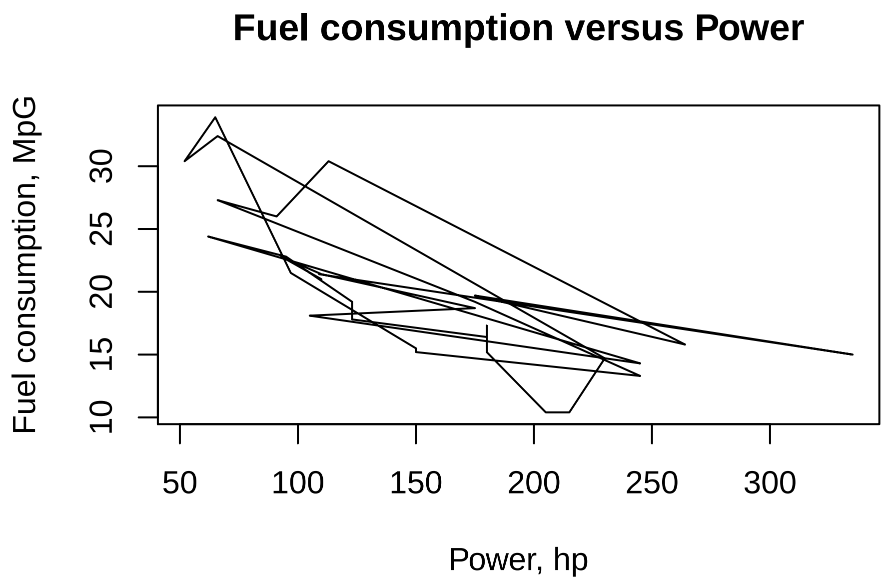


What does this graph show? There is a clear negative relationship between power and fuel consumption - higher power is associated with less miles per gallon, that is, high fuel consumption. However this relationship is not linear but somewhat curved.

Function `plot()` is a universal one and can be used for many different types of plots. Here are some examples.

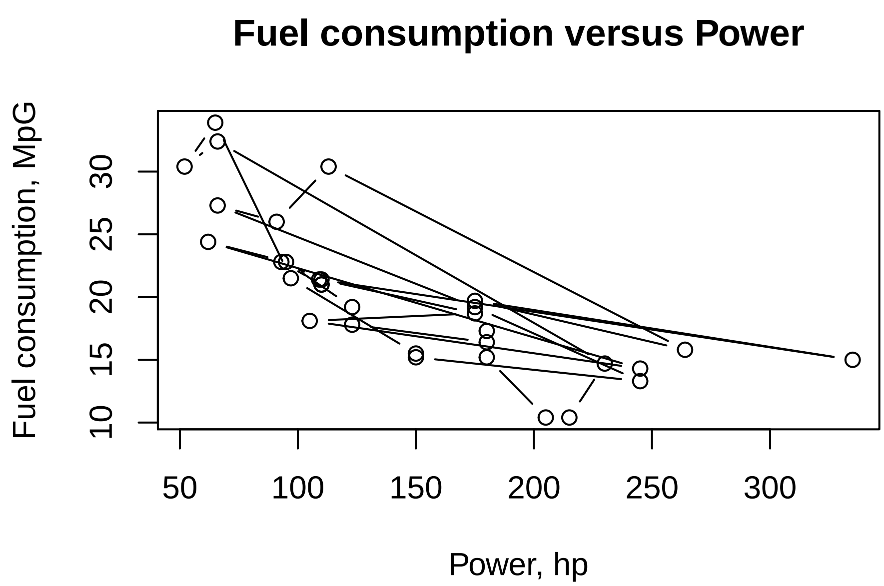
Lines chart (it does not make sense for these data, it is just an example)

```
plot(x=mtcars$hp, y=mtcars$mpg, type="l", main="Fuel consumption versus Power",
     xlab="Power, hp", ylab="Fuel consumption, MpG")
```



Combined chart It combines lines and points

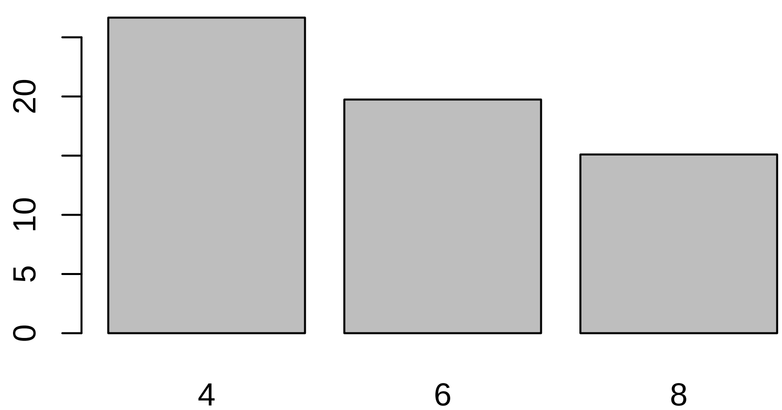
```
plot(x=mtcars$hp, y=mtcars$mpg, type="b", main="Fuel consumption versus Power",
     xlab="Power, hp", ylab="Fuel consumption, MpG")
```



Bar chart

Bar chart is used to study a relation between one numerical and one categorical variables. It is very important to think about aggregation function for numerical variable. Otherwise it is very easy to create a misleading data visualisation. It might be necessary to aggregate the data first and only after that try to plot it

```
# get average MPG per a number of cylinders by using tapply
temp_mpg_per_cylinder <- tapply(mtcars$mpg, mtcars$cyl, mean)
barplot(temp_mpg_per_cylinder)    # plot results
```



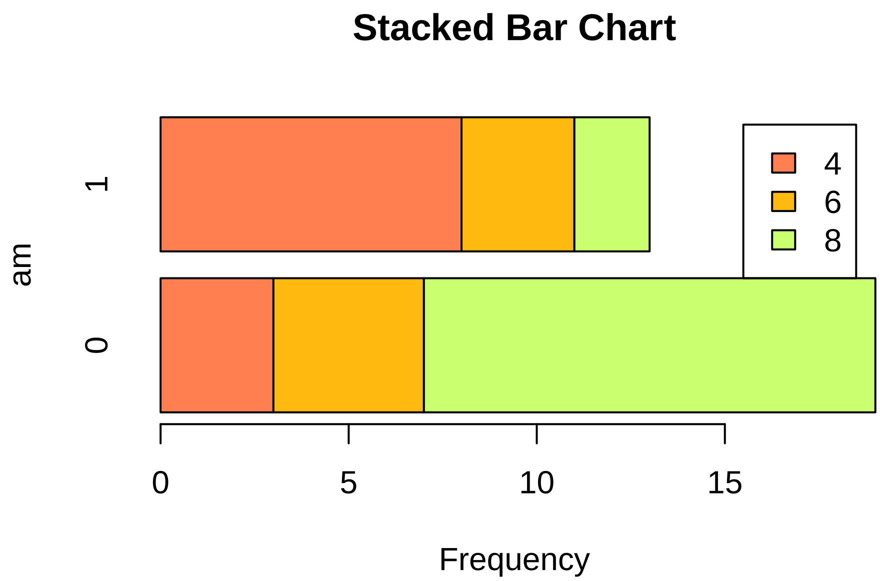
There are a lot of parameters in the `barplot()` function, however it works fine with default settings if you provide the right type of data for plotting.

It is possible to include two categorical variables in the bar chart. The result is a stacked bar chart or side-by-side bar chart. Again, it is necessary to prepare data for plotting first.

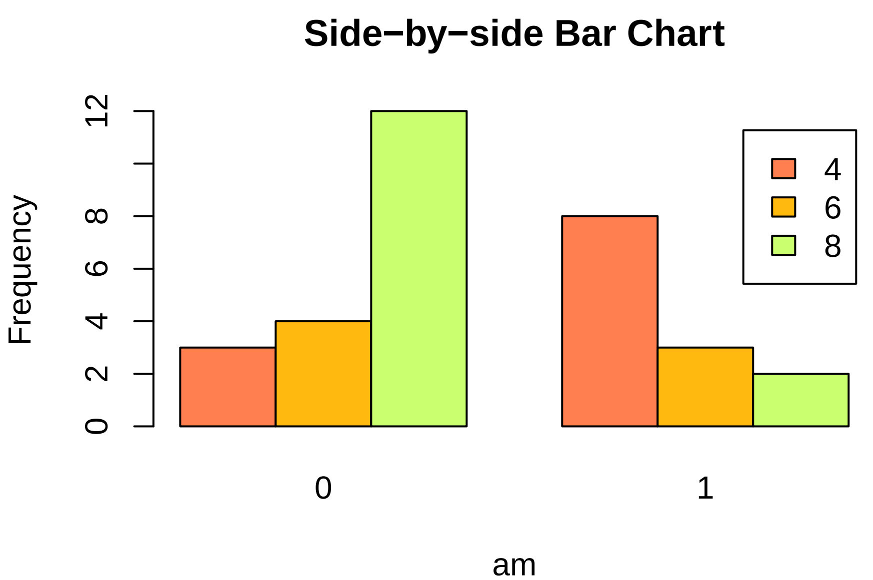
```
# get counts for cars by number of cylinders and transmission
counts <- table(mtcars$cyl, mtcars$am, dnn=c("cyl", "am"))
counts
```

```
##    am
## cyl  0  1
##   4  3  8
##   6  4  3
##   8 12  2
```

```
barplot(counts, main = "Stacked Bar Chart", xlab = "Frequency",
    ylab = "am", col = c("coral", "darkgoldenrod1", "darkolivegreen1"),
    legend = rownames(counts), horiz = TRUE)
```




```r
barplot(counts, main = "Side-by-side Bar Chart", xlab = "am",
    ylab = "Frequency", col = c("coral", "darkgoldenrod1", "darkolivegreen1"),
    legend = rownames(counts), beside = TRUE)
```

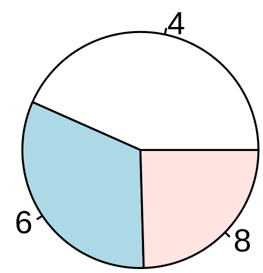


Pie chart

Pie chart is the most abused by mass media type of data visualisation. There are way to many example of wrong use of it. The pie chart can be used only if all components together create a whole object. For example, you have total sales for all companies and together they create a whole market. Hence, the pie chart will show a market share for each company.

Technically, pie chart is very close to a bar chart and the same type of data would be required. So, here is an example for the same data as above. Technically, it works. Practically, this is the wrong data visualisation for the data - bar chart is a better choice for this data.

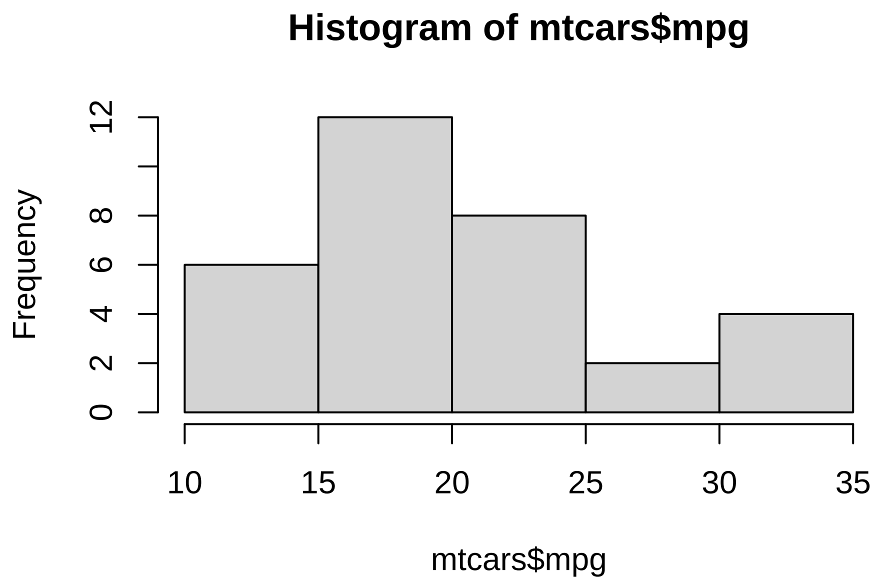
```
pie(temp_mpg_per_cylinder)
```



Histogram

Histogram is used to study a distribution of the numerical variable.

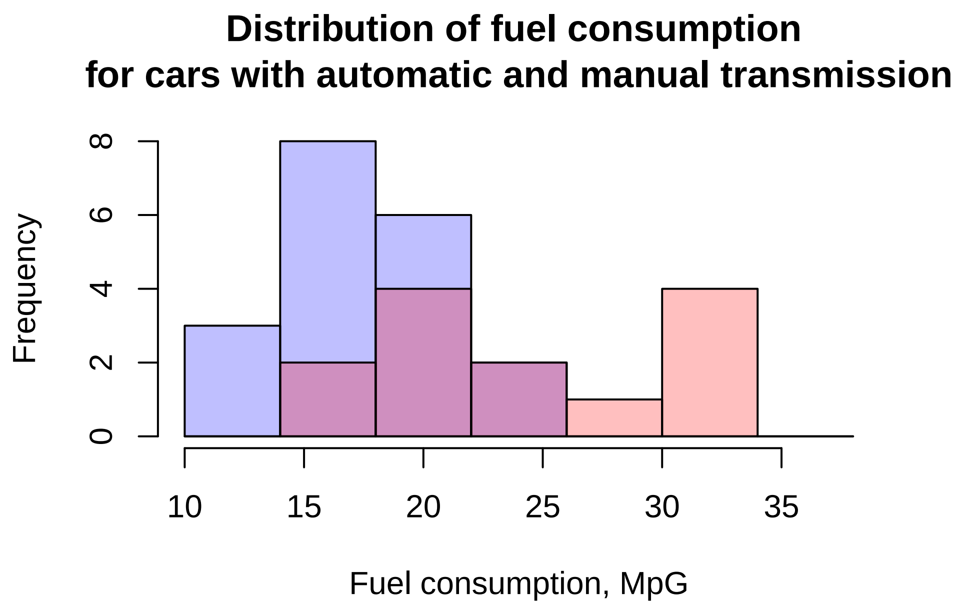
```
hist(mtcars$mpg)
```



It is possible to have multiple histograms on the same graph, so you can compare them. For this, you have to plot the first histogram and then add another one. The plot size is defined by the first graph, so it is important to adjust the size manually to have a spaces for the second graph. Size of the graph can be set by parameters `xlim` and `ylim`. Number of bars in the histogram is defined by parameters `breaks` – it can be used to "match" histograms and to set x axis limit.

For multiple histograms different colours and transparency levels should be set. First three numbers in the function `rgb()` set the colour in RGB scheme and the last one set transparency level at 25%. So, it is possible to see where two histograms overlap.

```
hist(mtcars$mpg[mtcars$am == 0], breaks=seq(10,40,4), col=rgb(0,0,1,1/4),
     main="Distribution of fuel consumption \nfor cars with automatic and manual transmission",
     xlab="Fuel consumption, MpG")
hist(mtcars$mpg[mtcars$am == 1], breaks=seq(10,40,4), col=rgb(1,0,0,1/4), add=TRUE)
```

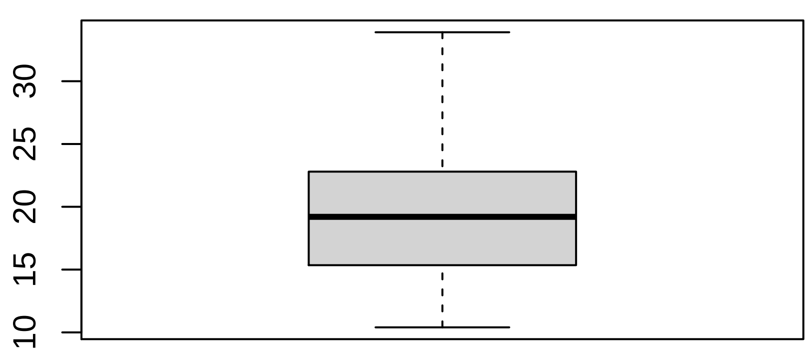


Size of the plotting area is set by the first histogram. Try to run the above code by plotting first the histogram for `am == 1`. You get "chopped" histogram and need to use parameter `ylim = 9` to fix the graph.

Box plot

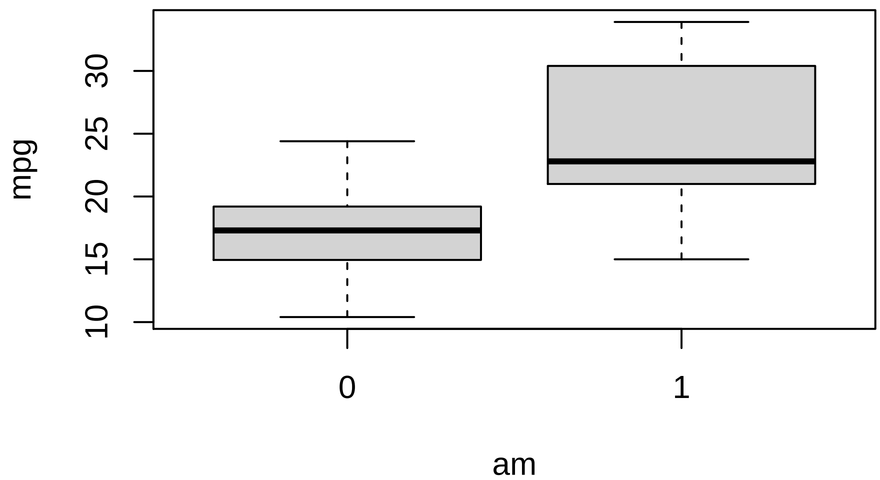
Box plot is another type of data visualisation to study a distribution of the numerical variable.

```
boxplot(mtcars$mpg)
```



It is possible to have multiple box plots side by side for easy comparison of distributions.

```
boxplot(mpg ~ am, data=mtcars)
```



Above command uses a so-called "formula" notation. Parameter `data` nominates a dataframe with the data. Formula `mpg ~ am` tells that variable `mpg` from the nominated dataframe depends on variable `am` from the same dataframe. So, you get two box plots - one for each category of car transmissions.

Formula notation is extremely popular in many R functions and you will see more examples shortly.

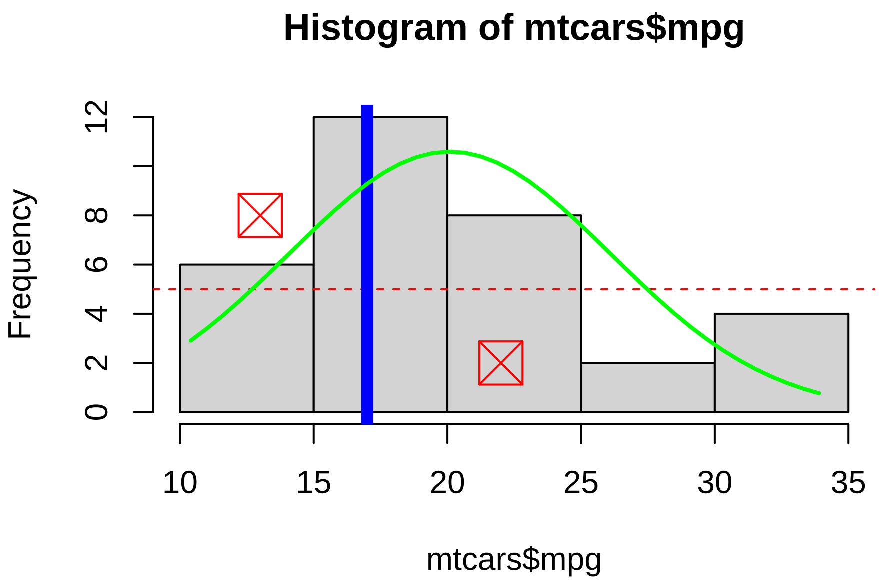
Adding extra elements

Extra lines or points can be added to any graph by using functions like `lines()`, `points()`, `abline()`, etc. These functions do not create new plotting window but add elements to the existing graph. Hence, you should have a plotting window available before using these functions.

```
hist(mtcars$mpg)                  # creates a plotting window and put a histogram there
abline(h=5, col="red", lty=2)     # add a horisontal line
abline(v=17, col="blue", lwd=6)  # add a vertical line
points(x=c(13, 22), y=c(8,2), pch=7, cex=3, col="red")     # add points

# prepare data for normal distribution density curve
xfit <- seq(min(mtcars$mpg), max(mtcars$mpg), length = 40)
yfit <- dnorm(xfit, mean = mean(mtcars$mpg), sd = sd(mtcars$mpg))
yfit <- yfit * length(mtcars$mpg) * 5

lines(xfit, yfit, col = "green", lwd = 2) # add a custom line
```



Multiple graphs

To have multiple data visualisations side by side, you have to prepare a space and then fill it with any graphs.

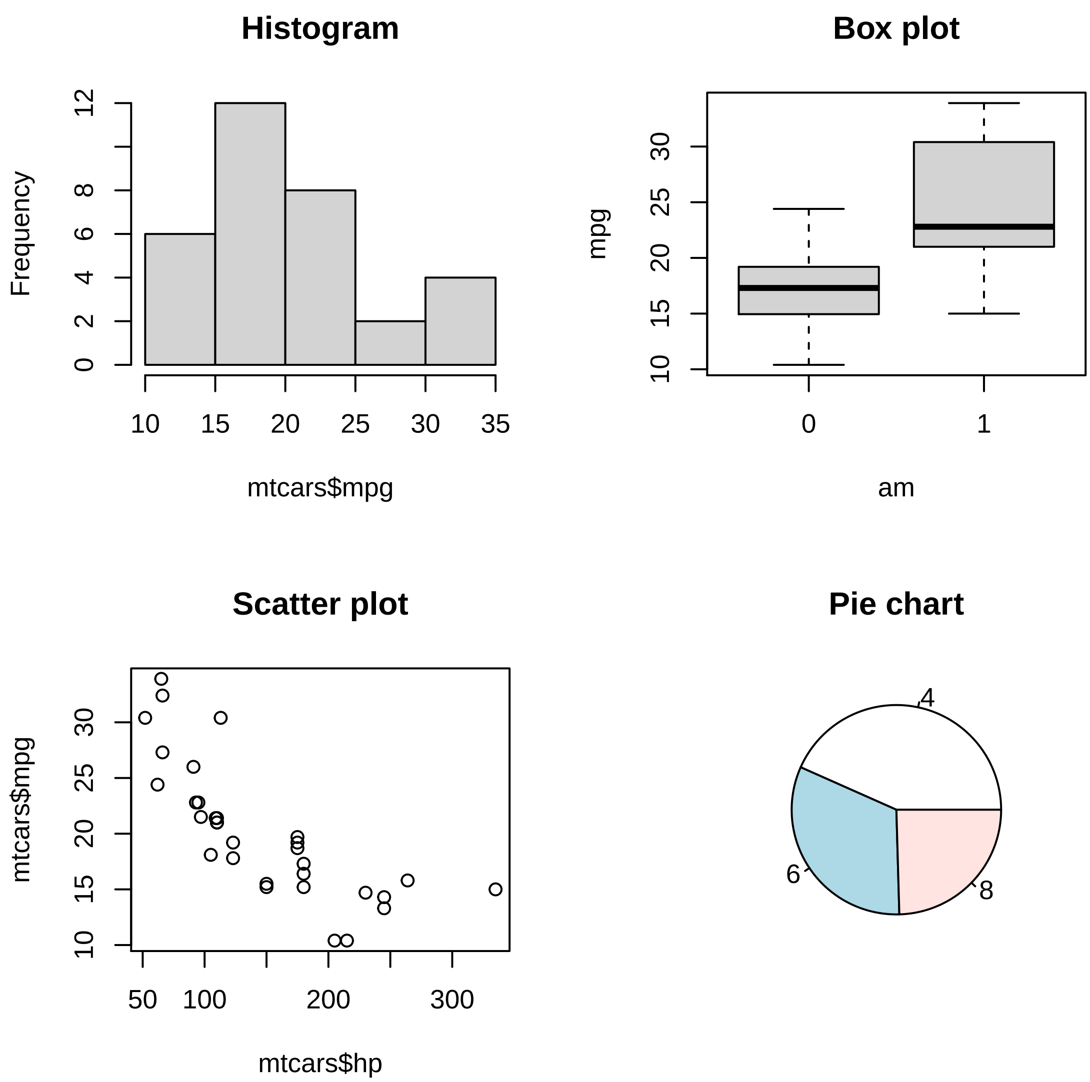
```
# create a graphics space that allows for 4 plots in total,
# 2 graphs in the each of 2 rows
par(mfrow = c(2, 2))

# create 4 plots
hist(mtcars$mpg, main = "Histogram")

boxplot(mpg ~ am, data=mtcars, main = "Box plot")

plot(x=mtcars$hp, y=mtcars$mpg, main = "Scatter plot")

pie(temp_mpg_per_cylinder, main = "Pie chart")
```




```r
# restore original graphic parameters
par(mfrow = c(1, 1))
```

Random numbers generation

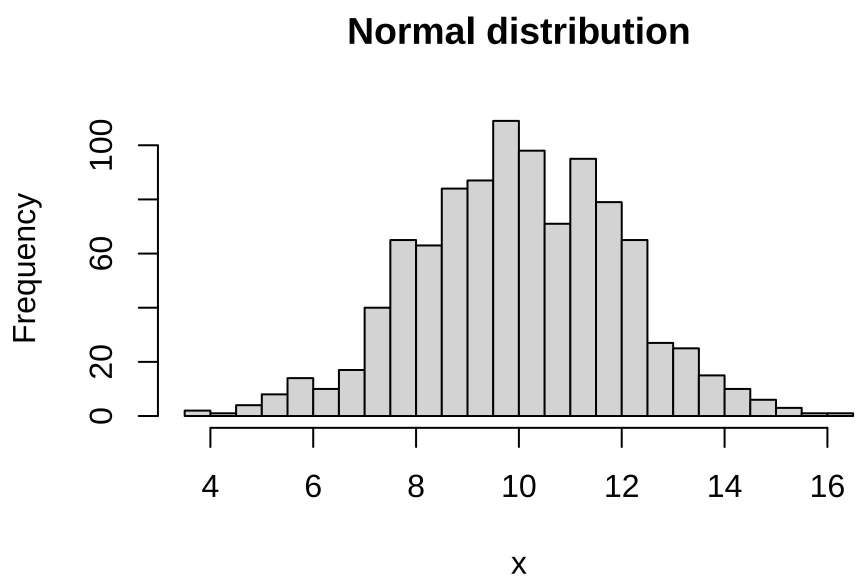
A very important functionality for any statistical package is a generation of random numbers. In reality, all computer generated random numbers are not really random. However, they are good enough for most practical purposes. Below are the most popular distributions.

Normal distribution

```
# 1000 numbers from Normal distribution with mean 10 and standard deviation 2
x <- rnorm(1000, 10, 2)
summary(x)                  # one more possible way to get summary statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.908   8.680   9.990  10.003  11.399  16.092
```

```
hist(x, breaks=20, main = "Normal distribution")
```

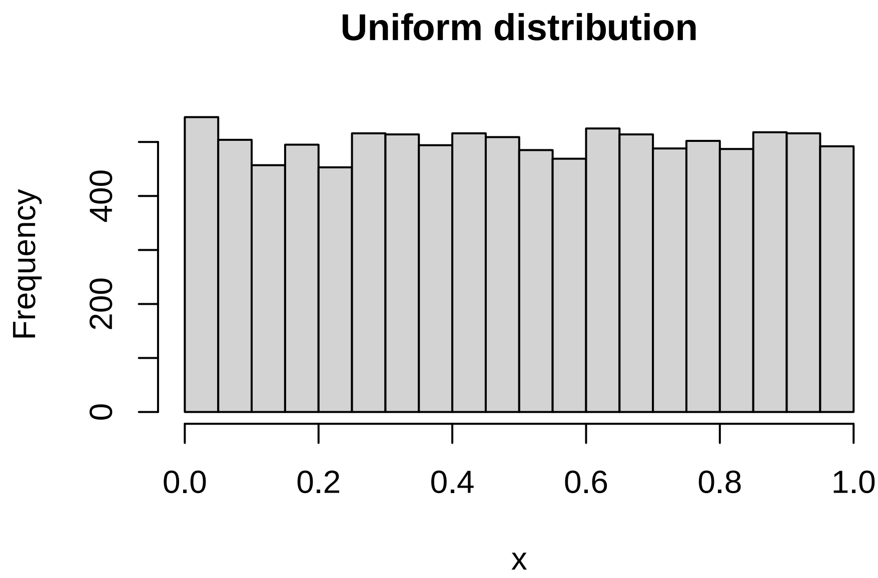


Uniform distribution

```
x <- runif(10000)           # 10,000 numbers uniformely distributed between 0 and 1
summary(x)                  # one more possible way to get summary statistics
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000234 0.2548926 0.4998599 0.5006991 0.7511137 0.9999837
```

```
hist(x, breaks=20, main = "Uniform distribution")
```

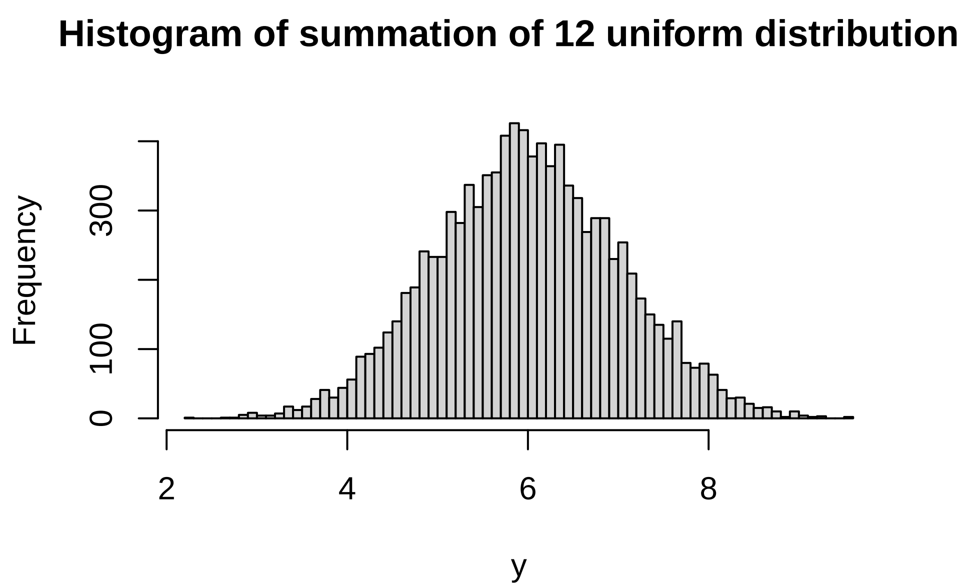


Here is an interesting illustration to a Center Limit Theorem - a summation of 12 uniform distributions

```r
# create 12 sets (columns) of 10000 uniformly distributed numbers each
x <- matrix(runif(120000), nrow=10000, ncol=12)

# sum up all 12 columns
y <- apply(x, 1, sum)

hist(y, breaks=100, main="Histogram of summation of 12 uniform distributions")
```



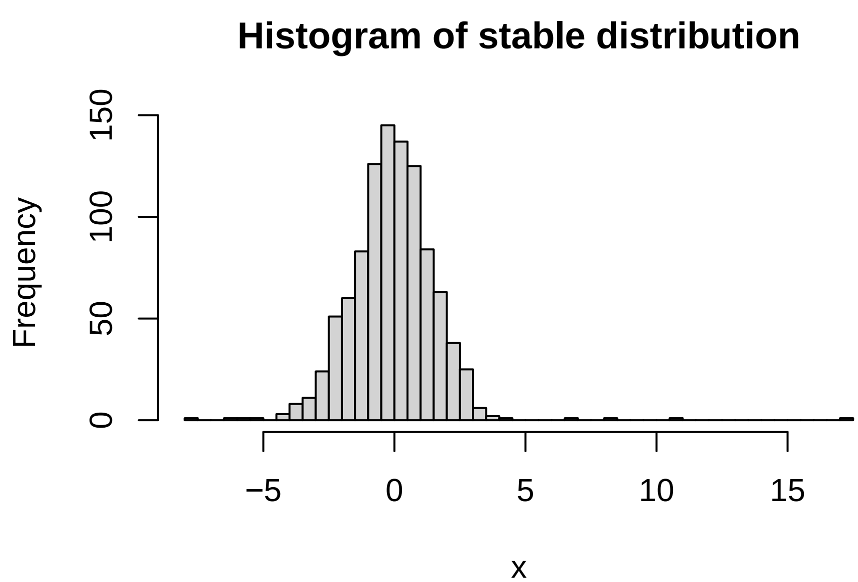
If you think that this is a normal distribution then it is not. However, it is very close to normal and it gets closer and closer as you increase a number of uniform distribution from 12 to infinity.

Other distributions

Try to type **?Distributions** to see a full list of distributions included in **stats** package loaded in R by default. Most distributions you might need are there.

If you need some other and more "exotic" distributions, you can try to look in other packages. For example, *stable* distribution can be found in packages **stabledist** and **stable**. The histogram might look like normal distribution, however it has longer tails.

```
x <- stable::rstable(1000, disp = 1, tail = 1.9)     # beware of the package used
hist(x, breaks=50, main="Histogram of stable distribution")
```



The full list of all packages providing different distributions can be found here: https://cran.r-project.org/web/views/Distributions.html

Statistical analysis – Statistical tests

Any type of statistical analysis you ever heard about is in R. Many tests do not require any packages as they are included in **stats** package loaded by default each time you start R. New or more complex tests and statistical procedures can be found in more than 16,000 packages available for R.

You had an Introductory statistics course or Research methods. You know statistics. Now you will see how to do the same job using R.

z-test

Z-test is (surprise-surprise!) not included in **stats** package. The reason is simple: it is not so popular. Z-test assumptions require a population standard deviation and it is never known to us. You can get Z-test from other packages, for example:

```
# This is a sample from a population with known standard deviation
x <- rnorm(12, mean=0, sd=2)
BSDA::z.test(x, mu = 1, sigma.x = 2, conf.level = 0.95)
```

```
##
##  One-sample z-Test
##
## data:  x
## z = -1.6353, p-value = 0.102
## alternative hypothesis: true mean is not equal to 1
## 95 percent confidence interval:
##  -1.075717  1.187454
## sample estimates:
## mean of x
## 0.0558687
```

Null hypothesis for the test was that mean equal 1. You know this is not true, as `x` was generated from the normal distribution with mean equal 0. Unfortunately, the sample size is so small that there are not enough evidences to reject the null hypothesis.

P-value is greater than significance level 5%. Hence, we fail to reject a null hypothesis - based on the data we have, the population mean is equal 1 or it can be equal 1.

95% confidence interval for the mean includes the hypothesised mean 1. Again, it is impossible to reject a null hypothesis.

Try to run the same code but for a larger sample and a story will be very different.

Proportions test

As mentioned before, Z-test is not very practical. Its main application in the real-life analysis is a proportions test. And R has a proportions test. Let's assume that a player played 10 games and won 6 games. Does it make them a really good player - someone who consistently wins more than 50% games? It might be possible that the player is not so good but just lucky.

```
prop.test(x=6, n=10, p=0.5, alternative = "greater", conf.level = 0.95)
```

```
##
##  1-sample proportions test with continuity correction
##
## data:  6 out of 10, null probability 0.5
## X-squared = 0.1, df = 1, p-value = 0.3759
## alternative hypothesis: true p is greater than 0.5
## 95 percent confidence interval:
##  0.3095345 1.0000000
## sample estimates:
##   p
## 0.6
```

P-value is greater than significance level 5%, hence there are not enough evidences to reject a null hypothesis and say that the player is really good.

Again, the sample size is extremely small. There are just 10 games. If there were more games and more wins then there could be another story. Try run the same code but for 60 wins out of 100 games.

t-test

T-test is a working horse of statistical analysis. Whenever there are just one or two samples to compare their means, there is a work for t-test. For example, re-call `mtcars` data set. Is fuel consumption for automatic transmission different to manual?

```
t.test(mpg ~ am, data = mtcars)
```

```
##
##  Welch Two Sample t-test
##
## data:  mpg by am
## t = -3.7671, df = 18.332, p-value = 0.001374
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -11.280194  -3.209684
## sample estimates:
## mean in group 0 mean in group 1
##        17.14737        24.39231
```

Function `t.test()` allows to specify samples `x` and `y` manually. However, formula notation as above is way more power and more convenient to use.

P-value is less than significance level 5%, hence there are enough empirical evidences to reject the null hypotheses about equal means between two groups of cars and conclude that means are not the same. Confidence interval for the difference shows you what group has a higher mean and on how much.

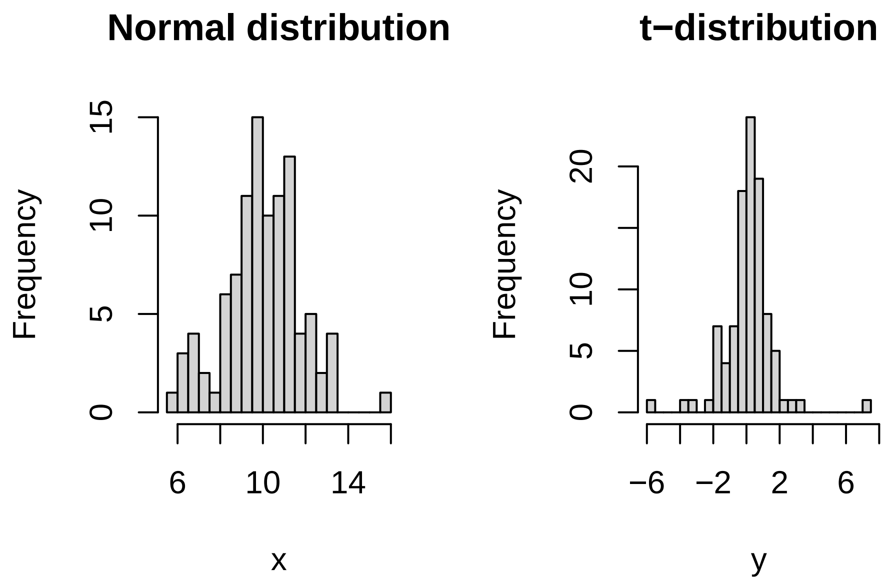
Normality test

An important assumption for many statistical tests is a normality of the data. There are several tests you can use to test for normality.

```
# generate data sets for an example
x <- rnorm(100, mean = 10, sd = 2)  # normal data
y <- rt(100, df=4) # t-distribution

# plot generated data
par(mfrow = c(1, 2))

hist(x, breaks=20, main = "Normal distribution")
hist(y, breaks=20, main = "t-distribution")
```




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

And now normality tests. All normality tests have the same null hypothesis: the data come from a normal distribution. If P-value is less than selected significance level, then we reject the null hypothesis and conclude that data are not normal.

Shapiro-Wilk test

```r
shapiro.test(x) # data look normal as it should be expected
```

```
##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.983, p-value = 0.2261
```

```r
shapiro.test(y) # data are not normal despite visual similarities in histograms
```

```
##
##  Shapiro-Wilk normality test
##
## data:  y
## W = 0.8825, p-value = 2.318e-07
```

Kolmogorov-Smirnov test

```r
ks.test(x, "pnorm", mean = mean(x), sd = sd(x)) # again data look normal
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  x
## D = 0.06638, p-value = 0.7703
## alternative hypothesis: two-sided
```

```r
ks.test(y, "pnorm", mean = mean(y), sd = sd(y)) # this data look normal too
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  y
## D = 0.12182, p-value = 0.1028
## alternative hypothesis: two-sided
```

Not all test are the same and some tests are more or less sensitive than others.

Anderson-Darling test

```r
# beware of an extra package

nortest::ad.test(x) # again data look normal
```

```
##
##  Anderson-Darling normality test
##
## data:  x
## A = 0.5007, p-value = 0.2034
```

```r
nortest::ad.test(y) # and non-normal again
```

```
##
##  Anderson-Darling normality test
##
## data:  y
## A = 2.653, p-value = 9.796e-07
```

ANOVA test

Analysis of variance or ANOVA test is used for situations when there are more than two groups, so t-test can not be used. This is an extremely powerful test and it can be used in many different ways. The most simple example is to use `mtcars` data set again and check fuel consumption for three groups of cars with 4, 6 and 8 cylinders. You know from the descriptive statistics and data visualisation presented before that fuel consumption was different. However, is this difference statistically significant?

```r
# column "cyl" should be treated as categorical variable
# hence, we convert it to factor
model.fit <- aov(mpg ~ as.factor(cyl), data = mtcars)
summary(model.fit)
```

```
##                Df Sum Sq Mean Sq F value   Pr(>F)    
## as.factor(cyl)  2  824.8   412.4    39.7 4.98e-09 ***
## Residuals      29  301.3    10.4                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

A null hypothesis for ANOVA is that there is no difference between groups. In the test results, P-value is almost zero, so the model is statistically significant. We can reject the null hypothesis and conclude that not all means are the same. At least one group is different to some others and this difference is statistically significant.

ANOVA test would not tell what group has mean different to other groups and how many such "different" groups. To find out such details, you need a post hoc test – Tukey test. Tukey test will study every possible pair of groups.

```r
TukeyHSD(model.fit, conf.level = 0.95)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
## 
## Fit: aov(formula = mpg ~ as.factor(cyl), data = mtcars)
## 
## $`as.factor(cyl)`
##           diff        lwr        upr     p adj
## 6-4  -6.920779 -10.769350 -3.0722086 0.0003424
## 8-4 -11.563636 -14.770779 -8.3564942 0.0000000
## 8-6  -4.642857  -8.327583 -0.9581313 0.0112287
```

As you can see, all P-values are less than significance level 5%, hence we have to reject all null hypothesises (there are 3 null hypothesises) that there are no differences between groups. We conclude that every group is different to every other group. Confidence intervals for differences show direction for these differences: an average value of miles per gallon (mpg) (1) for 6 cylinder car is less than for 4 cylinder cars; (2) for 8 cylinder cars is less than for 4 cylinder cars; and (3) for 8 cylinder cars is less than for 6 cylinder cars.

If your research requires higher level of confidence, for example you need 99% confidence. Then P-value for the difference between 8 and 6 cylinder cars would be not statistically significant and you will not be able to say that there is a difference in fuel consumption between these two groups. Try to change confidence level and run above code again.

Correlation

Correlation is a measure of linear relationship between two or more numerical variables. There are two functions in `stats` that can calculate correlation `cor()` and `cor.test()`. Both of them can calculate Pearson, Kendall or Spearman correlations depending on what choice you made for your data. The first function gives you just a coefficient of correlation. The second one runs a formal test with a null hypothesis that correlation equals zero.

Let's have a look on the relationship between miles per gallon and horse power in `mtcars` data set.

```r
cor(x = mtcars$mpg, y = mtcars$hp, method = "pearson")
```

```
## [1] -0.7761684
```

There is a negative correlation as you should remember from the scatter plot before. However, is it statistically significant?

```r
cor.test(x = mtcars$mpg, y = mtcars$hp, method = "pearson", conf.level = 0.95)
```

```
##
##  Pearson's product-moment correlation
##
## data:  mtcars$mpg and mtcars$hp
## t = -6.7424, df = 30, p-value = 1.788e-07
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.8852686 -0.5860994
## sample estimates:
##        cor
## -0.7761684
```

P-value is less than significance level 5%, hence we have enough empirical evidences to reject a null hypothesis that correlation is zero and conclude that correlation is not zero. That means, there is a relationship between fuel consumption and power of the car.

95% confidence interval shows that the correlation coefficient is somewhere between -0.59 and -0.88.

Linear regression

Leaner regression analysis builds a linear model of a relationship between the target and predictor. The purpose of the analysis is double-folded: (1) make a prediction about new observations; (2) study an effect of a predictor(s) on the target. In the below example based on the `mtcars` data set, miles per gallon is a target and horse power is a predictor. It is known from the previous analysis that there is a very strong correlation between them, so the model should be good

```r
lm.model <- lm(mpg ~ hp, data = mtcars)
summary(lm.model)
```

```
##
## Call:
## lm(formula = mpg ~ hp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.7121 -2.1122 -0.8854  1.5819  8.2360
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 30.09886    1.63392  18.421  < 2e-16 ***
## hp          -0.06823    0.01012  -6.742 1.79e-07 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.863 on 30 degrees of freedom
## Multiple R-squared:  0.6024, Adjusted R-squared:  0.5892
## F-statistic: 45.46 on 1 and 30 DF,  p-value: 1.788e-07
```

Overall model P-value is less than 5% and R-squared is about 59%, which is not too bad.

Both intercept and slope are statistically significant as individual P-values are less than significance level 5%. An interpretation for a slope follows: every extra unit of "horse power" takes away in average 0.07 miles per gallon of fuel consumption. An interpretation for an intercepts is: a car with zero horse power has fuel consumption of 30 miles per gallon. It does not make much sense but that is OK, it happens. An intercept interpretation might be meaningless. A slope interpretation is always meaningful.

Non-parametric tests

All previous tests had an assumption that data are normally distributed. If your data are not normal than you should consider non-parametric tests.

Null hypothesis and result interpretations are similar to the parametric tests reviewed before.

Wilcoxon test or Mann-Whitney test

This test works for one or two samples or groups

```r
wilcox.test(mpg ~ am, data = mtcars)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  mpg by am
## W = 42, p-value = 0.001871
## alternative hypothesis: true location shift is not equal to 0
```

Kruskal-Wallis rank sum test

This test can be used for two or more groups

```r
kruskal.test(mpg ~ cyl, data = mtcars)
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  mpg by cyl
## Kruskal-Wallis chi-squared = 25.746, df = 2, p-value = 2.566e-06
```