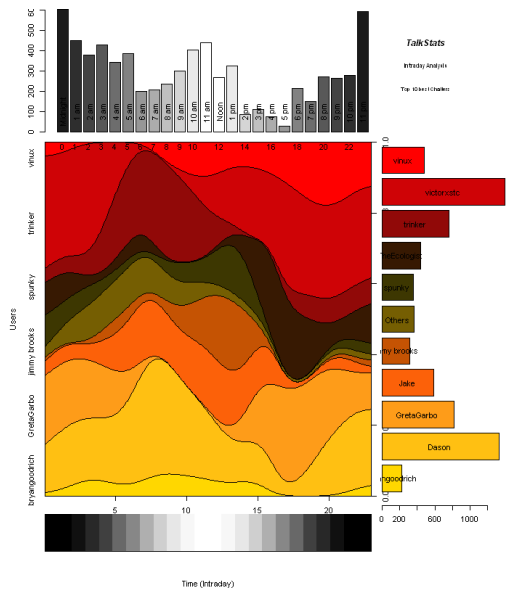
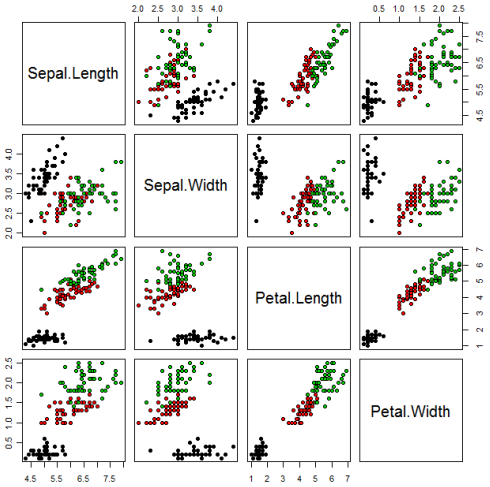
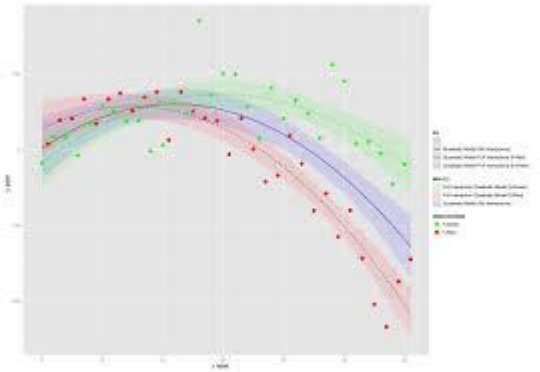
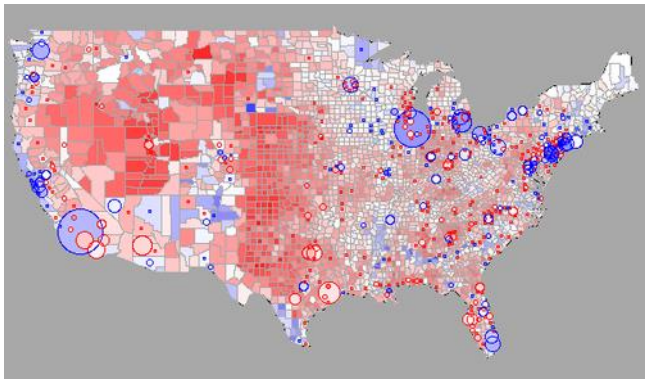
Introduction to R

Dr. Ron Rotkopf (ron.rotkopf@weizmann.ac.il)
Bioinformatics Unit, Life Sciences Core Facilities









What is R?

- Scripting language
- Free
- Open-source
- Runs on all popular platforms (Windows, Mac, Linux)
- Large user community
- Widely used for statistical computing and graphics
- Many extra functions via packages

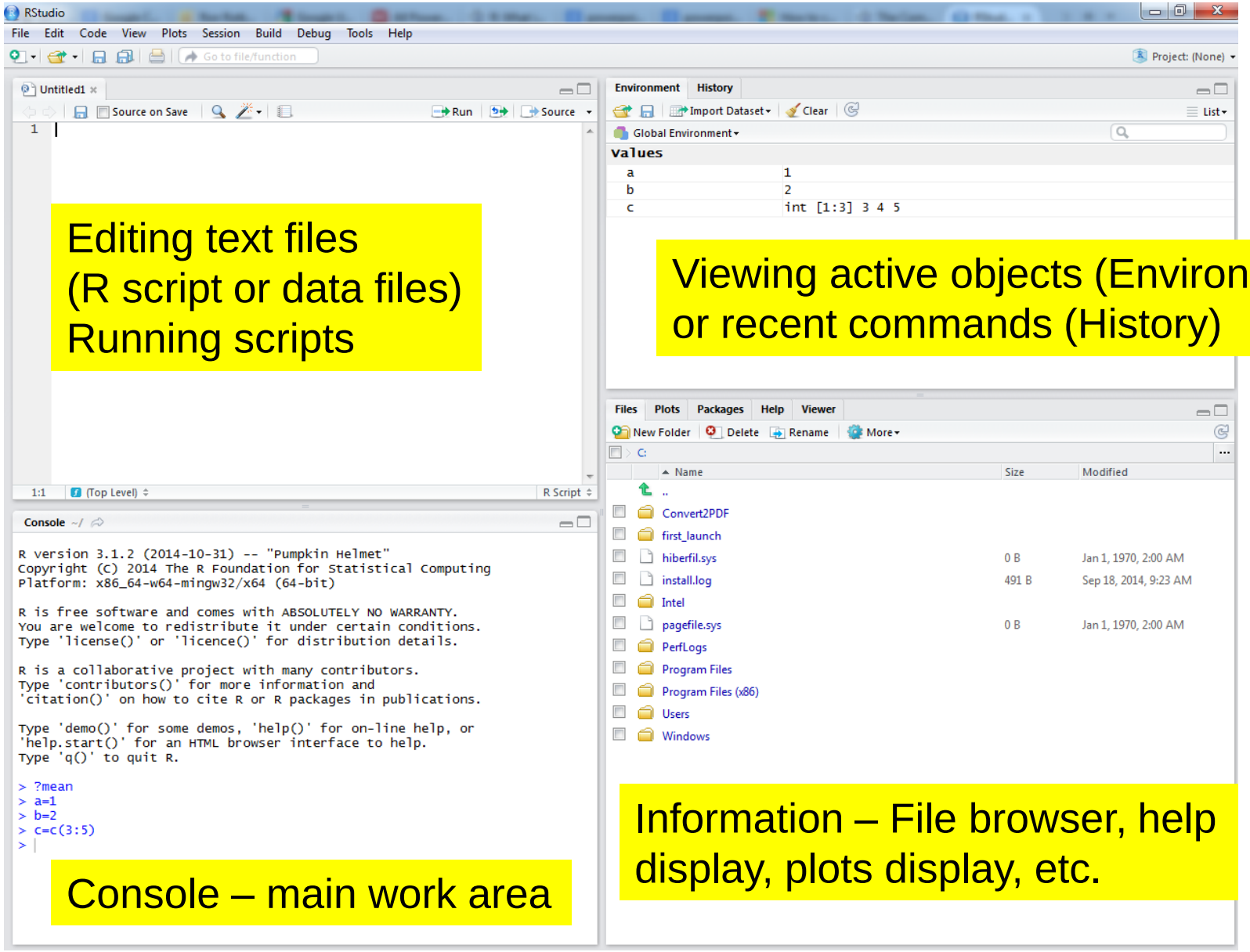
Installing R and RStudio

R:
http://cran.rstudio.com/

RStudio:
http://www.rstudio.com/products/rstudio/download/

via Wexac:
http://appsrv.wexac.weizmann.ac.il/rstudio/

The RStudio Interface



Entering commands

- From the console:

“Enter” to run a command.

Up arrow to access recently-entered commands.

Tab to fill in functions or variable names.

- From the text editor:

Ctrl+Enter to run one line or selection.

Ctrl+Shift+Enter to run entire script.

If you want to write comments or “mute” a specific line, use #.

Each command should be written in a new line - Several commands on the same line can be separated with ;

Data types

- Everything is case-specific! Use letters, numbers and periods for object names.

- Assigning a single value: `a<-5`
  `a=5`

- "`<-`" and "`=`" are the same: both assign values to the object on the left. Shortcut for "`<-`" is "Alt –"

- Multiple values (vector): `a=c(1,3,5,7)` Specific values
  `b=c(1:100)` Ascending sequence
  `d=rep(0,50)` Repeat 0 fifty times

Help for any function: `?function.name`

Example: `?sum` `?seq` `?mean`

More general search: `??search.string`

Note that when copying from Office to R, parentheses may need to be re-typed.

- A vector can contain one data type: numeric, character or logical.
- numeric: `a=c(4.5,3.14,5.2,6.8)`
- character: `b=c("Bob","Alice","Jack","Jill")`
- logical: `d=c(TRUE,FALSE,TRUE,TRUE)`

`TRUE` can also be entered as `T` or `1`
Special case - `NA`

- Data type will be presented in the "Environment" window.
- You can check data type with "`is`":

```
is.numeric(varname)
is.character(varname)
is.logical(varname)
is.na(varname)
```

You can change data type with "as":

```
as.numeric(varname)
as.character(varname)
as.logical(varname)
```

Calling a specific cell or cells – square brackets:

```
a[5]
a[c(5,7,9)]
```

multiple values should always be connected with `c()`

Calling everything except one cell:

```
a[-5]
```

The required indices can come from another variable (numeric or logical).

Example:

```
a=c(21:30)
b=c(2,4,6)
d=c(F,T,F,T,F,T,F,F,F,F)
```

`a[b]` and `a[d]` will give the same results.

Logical vector (True-False)

Can be used as a filter by comparing to another vector or single value.

`logvec=a=="Bob"` mark as TRUE cells containing "Bob" (character comparison)
`logvec=a>5` mark as TRUE cells larger than 5 (numeric comparison)

Possible comparisons:
==
>
<
>=
<=

Combinations:
! NOT
& AND
| OR

Note that "=" is for assigning values, "==" is for comparing values.

Example of using logical vectors:

X is a numeric vector.
`mean(X)` and `sum(X)` will calculate the mean and sum for X.

`sum(X>25)` will count how many numbers in X are bigger than 25.

`mean(X>25)` will calculate the proportion of numbers in X that are bigger than 25.

This is because `X>25` creates a logical vector, transformed to values of 0 (FALSE) and 1 (TRUE).

X	21	22	23	24	25	26	27	28	29	30
X>25	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE
	0	0	0	0	0	1	1	1	1	1

Matrices

Tables – containing rows and columns.
All cells must be of the same type (numeric, character, etc.)

Generating a new matrix:
`y=matrix(1:20, nrow=5,ncol=4)`

A new matrix can also be filled with zeroes or NAs.

Accessing specific cells is done by row number and column number:

```
y[,4] # 4th column of matrix
y[3,] # 3rd row of matrix
y[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```

Naming rows: `rownames(y)=c("P1","P2","P3","P4","P5")`
naming columns: `colnames(y)=c("height","weight","bp","chol")`

Connecting matrices:
`mat3=cbind(mat1,mat2)` connects by columns – one next to the other.
`mat4=rbind(mat1,mat2)` connects by rows – one over the other.

Data frames

Very similar to matrices, but can contain different data types in each column.
A data frame can be created:

- by connecting vectors.
- by transforming a matrix.
- by reading from a text file.

Connecting vectors:

```
d=c(1,2,3,4)
e=c("red", "white", "red", NA)
f=c(TRUE,TRUE,TRUE,FALSE)
mydata=data.frame(d,e,f)
names(mydata)=c("ID","Color","Passed")
```

Transforming a matrix:

```
mat1=matrix(1:20,5,4)
dat1=as.data.frame(mat1)
```

Reading from a file:

```
dat1=read.csv("filename.csv")
```

"csv" is a comma-separated text file, which can be

saved and viewed from Excel.

Options for other files (e.g. tab-separated) are `read.table` or `read.delim` –

see the ?read.table help page for options.

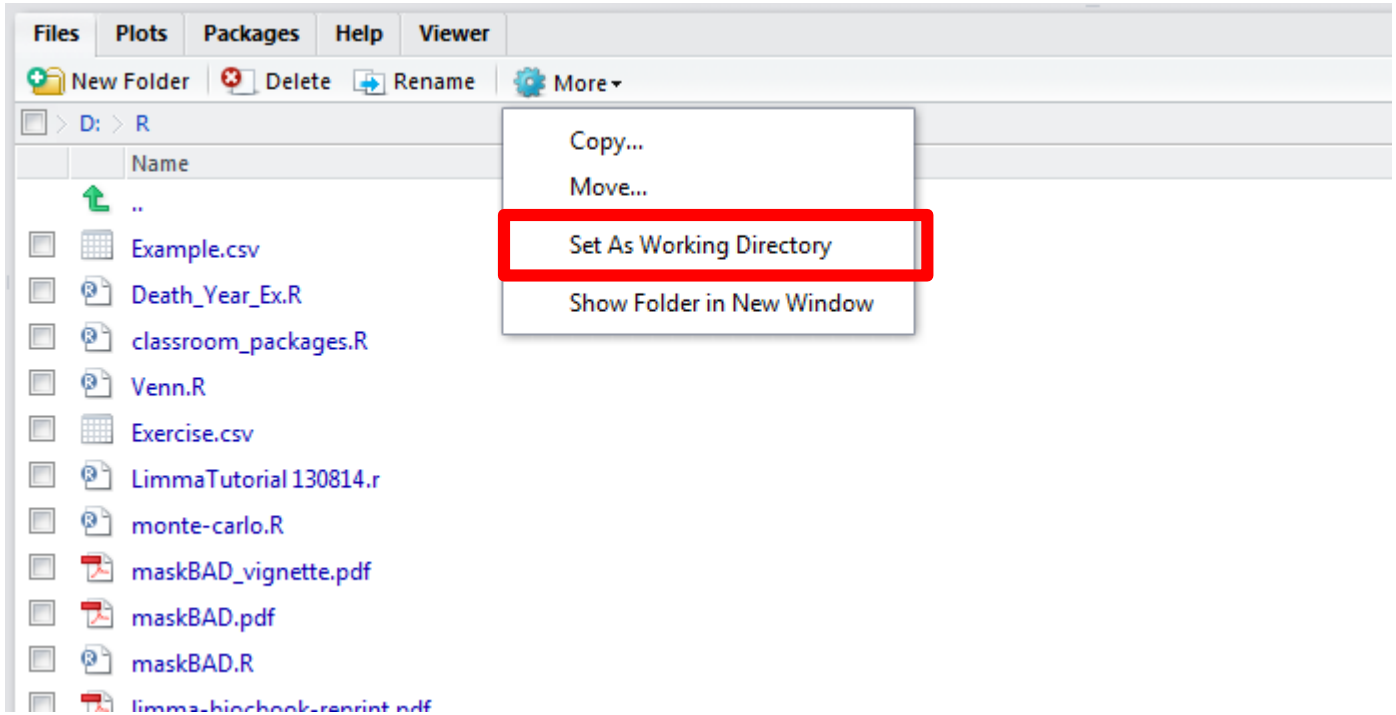
The file location can be typed with the full path

or by first setting the working directory with `setwd()`.

Tables can also be imported via "Import Dataset" in RStudio.

You can write data frames to a file using `write.csv(dfname, "filename.csv")`

Setting the working directory:
`setwd("full_path")` or through the menu:



When preparing your data in Excel:

- Keep only the data table – no graphs or comments, no empty lines or columns.
- If a column is numeric, it can't contain any comments, question marks, etc.
- If a column indicates groups, make sure that they are marked uniformly, accounting for case-sensitivity (i.e. control vs. Control)
- For missing data just leave empty cells – they will be converted to NA by R.
- Column names will be used as variable names, so they should not contain special characters – the safest way is to use only letters, numbers, and periods for separation (e.g. `night.blood.pressure1`)
- When all is ready, save as csv file (comma-delimited).

Accessing data frame elements

- By index number (like in matrices):
  ```
  myframe[3:5] # columns 3,4,5 of data frame
  ```
  Pay attention to whether you're calling rows or columns!
  With no comma, R assumes you mean columns.

- ```
  By column names:
  myframe[c("ID","Age")] # columns ID and Age from data
  frame
  ```

- ```
  By column names with $ separator:
  myframe$ID # variable ID in the data frame
  ```

Lists

A list is a “collection” of different types of variables.
We won’t have much use for creating lists ourselves,
but they are usually the output of more complex functions.

```
w=list(name="Fred", mynumbers=a, mymatrix=y, age=5.3)
```

character | numeric vector | matrix | numeric

A list can also contain several smaller lists:

```
v=c(list1,list2)
```

Components of a list can be accessed using index numbers or variable names:

```
mylist[[2]] # 2nd component of the list
mylist[["mynumbers"]] # component named mynumbers in list
mylist$mynumbers   # same as previous row
```

Factors

If a column in our data indicates groups, and not individual levels,
then it should be defined as a factor, and not a character vector.
This is usually done automatically when importing a data frame.

```
data$Treatment = as.factor(data$Treatment)
```

This identifies the unique values in the vector,
and assigns numbers instead.

Ways to avoid this:

while importing: `dat1=read.csv("filename.csv", stringsAsFactors=FALSE)`

on an existing table: `data$Treatment = as.character(data$Treatment)`

Calling Subsets

Calling only specific rows (or columns) to a new object, based on a given condition.

```
newdata=subset(olddata, age >= 20 | age < 10)

Alternate:
newdata=olddata[olddata$age>=20 | olddata$age<10,]
```

Attachment of data frame

`attach(olddata)`
Enables accessing specific columns without using the data frame name repeatedly.
Note that changing values in the attached columns does not change the original values in the table!

```
newdata=olddata[age>=20 | age<10,]
```

When you're done with a table, use `detach.`

More useful functions

```
length(object) # number of elements or components
str(object)    # structure of an object
class(object)  # class or type of an object
names(object)  # column names of a data frame
nrow(object) # number of rows of a data frame
head(object) # presents the first 6 rows of an object
tail(object) # presents the last 6 rows of an object

ls()       # list current objects
rm(object) # delete an object
```

Control structures

If statements

```
if (logical condition) {
    command1
     command2
     …
} else {
    command3
    command4
… }
```

Note the use of curly brackets for multiple commands.
The “else” part is optional.

“For” loop:

Repeat through the following commands a specified number of times.

```
for (var in seq) {
   command1
   command2
}
```

“var” is a counter variable - i and j are commonly used, but you can use any name you like.

“seq” are the numbers (or other values) to go through – can be predefined, e.g. 1:10, or related to the length of a vector, e.g. 4:length(x))

“If” and “For” Example

```
dat=runif(20)  #generates 20 random numbers between 0 and 1
for (i in 1:20) {
  if (dat[i]<0.5)
    dat[i]=0
}
```

Loops can many times be avoided by using operations on entire columns/vectors.

```
dat=runif(20)
dat[dat<0.5]=0  # accomplishes the same as the loop
```

Installing a package from CRAN

CRAN - **The Comprehensive R Archive Network**

`install.packages("package.name")` – done only once per installation

`library(package.name)` – done once per session

For Bioconductor packages, the syntax is different, e.g.:

```
source("https://bioconductor.org/biocLite.R")
biocLite("limma")
library(limma)
```

melt (from ‘reshape’ package)

Converts a data frame from ‘wide’ form to ‘long’ form

```
> wide.df
   Group1 Group2 Group3
1       1     11     21
2       2     12     22
3       3     13     23
4       4     14     24
5       5     15     25
6       6     16     NA
7       7     17     NA
8       8     18     NA
9       9     19     NA
10     10     20     NA
```

```
long.df=melt(wide.df)
```

```
> long.df
   variable value ID
1    Group1     1  1
2    Group1     2  2
3    Group1     3  3
4    Group1     4  4
5    Group1     5  5
6    Group1     6  6
7    Group1     7  7
8    Group1     8  8
9    Group1     9  9
10   Group1    10 10
11   Group2    11 11
12   Group2    12 12
13   Group2    13 13
14   Group2    14 14
15   Group2    15 15
16   Group2    16 16
17   Group2    17 17
18   Group2    18 18
19   Group2    19 19
20   Group2    20 20
21   Group3    21 21
22   Group3    22 22
23   Group3    23 23
24   Group3    24 24
25   Group3    25 25
26   Group3    NA 26
27   Group3    NA 27
28   Group3    NA 28
29   Group3    NA 29
30   Group3    NA 30
```

merge

Merges two data frames based on a common column.

```
merge(age.df, kids.df, by="Name")
```

```
merge(age.df, kids.df, by="Name“, all=TRUE)
```

```
> age.df
     Name Age
1  Arthur  23
2     Bob  45
3 Charlie  67
4   David  54
5   Ethan  35
```

```
> kids.df
     Name children
1   Ethan        2
2 Charlie        4
3     Bob        3
4   Garry        1
5  Arthur        0
```

```
> merge(age.df, kids.df, by="Name")
     Name Age children
1  Arthur  23        0
2     Bob  45        3
3 Charlie  67        4
4   Ethan  35        2
```

```
> merge(age.df, kids.df, by="Name", all=TRUE)
     Name Age children
1  Arthur  23        0
2     Bob  45        3
3 Charlie  67        4
4   David  54       NA
5   Ethan  35        2
6   Garry  NA        1
```

aggregate

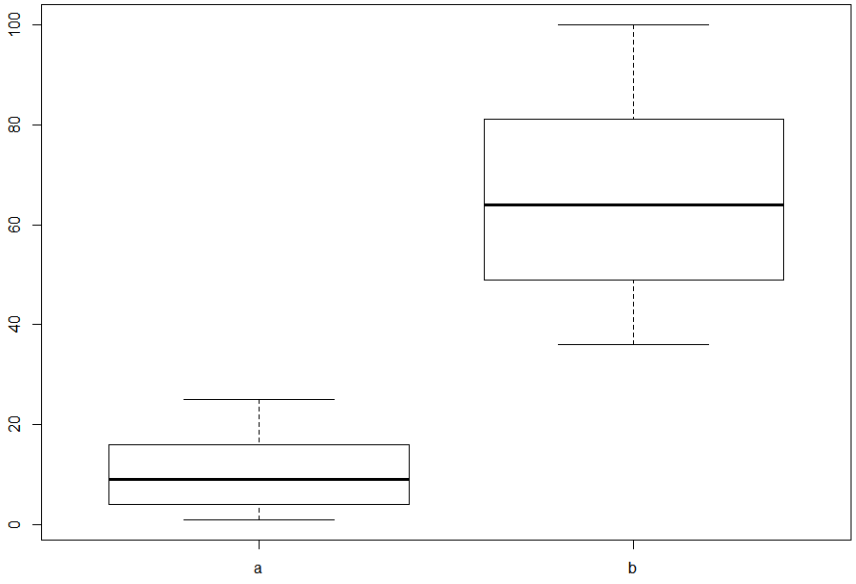
Runs a chosen function on subsets of the data frame, and collects the results.

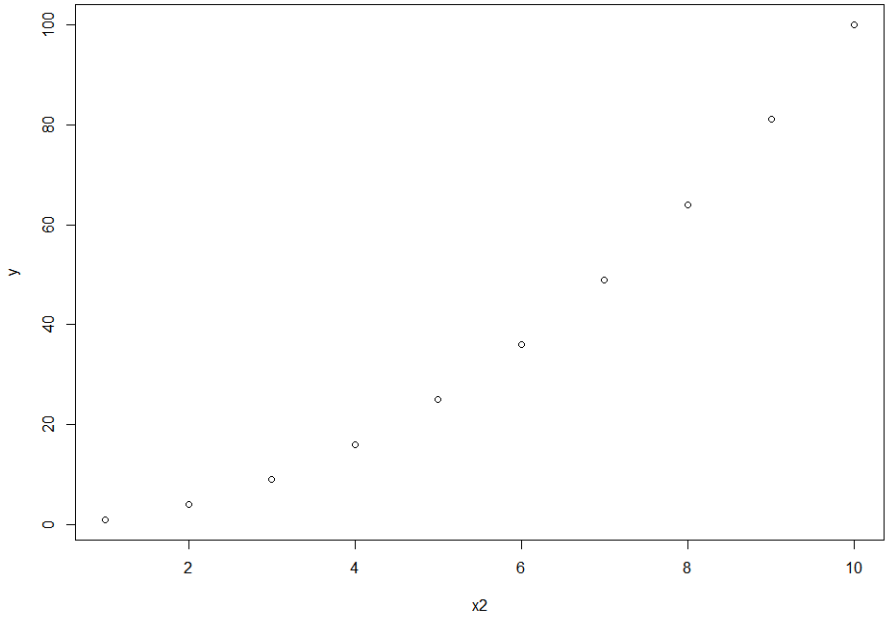
```
averages = aggregate(formula = mpg~gear, data = mtcars, FUN=mean)
```

Calculates the mean mpg for each level of “gear”.

The “plot” function

`plot(x,y)` plots y as a function of x.
Can result in a scatterplot or a boxplot, depending on the type of data.





“plot” - more options

`type`

what type of plot should be drawn. Possible types are
"p" for **p**oints, "l" for **l**ines,
"b" for **b**oth, "c" for the lines part alone of "b",
"o" for both ‘**o**verplotted’, "h" for ‘**h**istogram’ like (or ‘high-density’) vertical lines,
"s" or “S” for st**a**ir **s**teps.

`main`	overall title for the plot
`sub`	sub title for the plot
`xlab`	title for the x axis
`ylab`	title for the y axis

Example:

```
plot(orig, squared, type = “o”,
main = “Squared over original values”,
xlab = “Original”, ylab = “Squared”)
```

There are plenty of more options for changing the graph appearance, here are only a few of them:

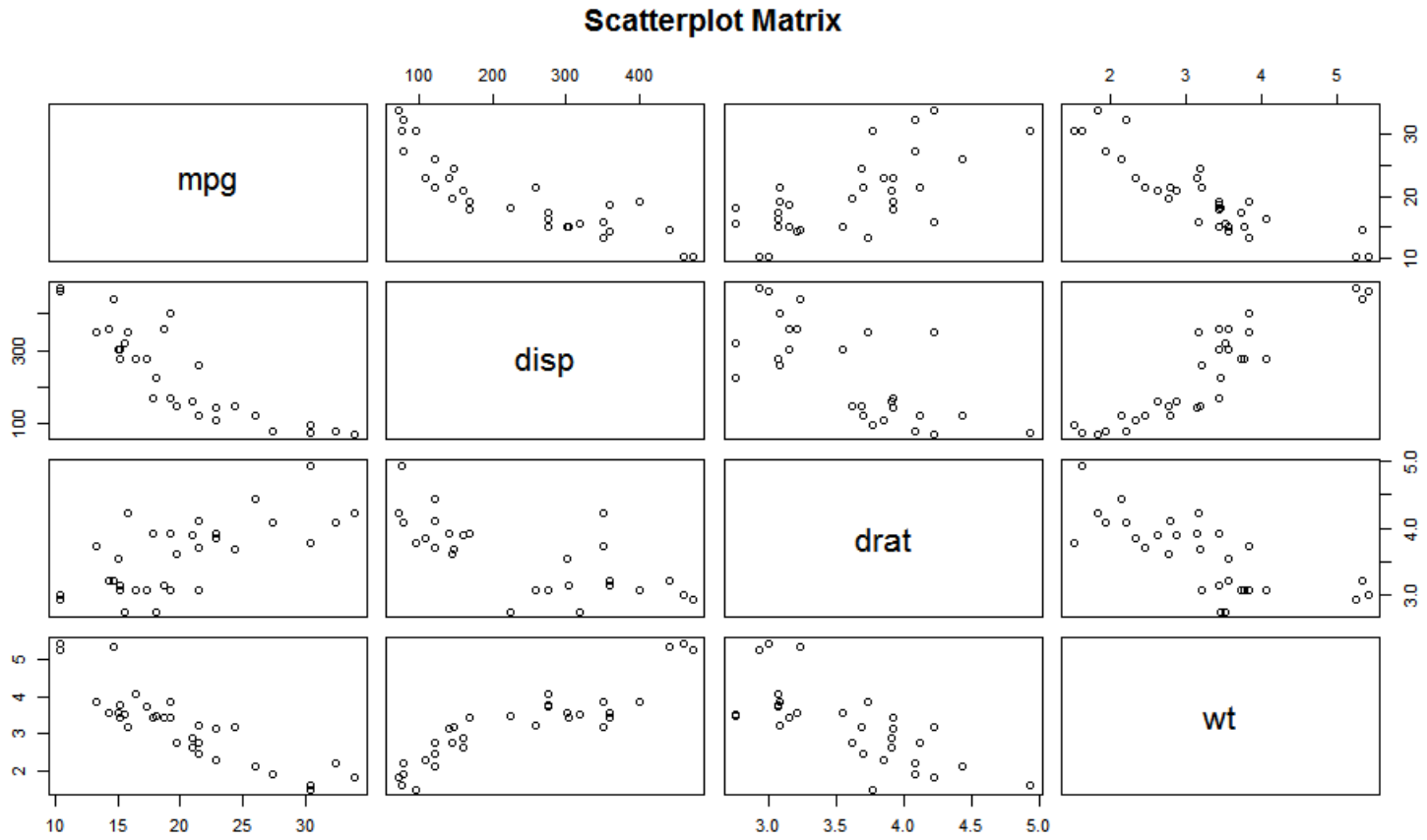
`cex` Change size of text and symbols
`pch` Change symbol type for points
`lty, lwd` Change line type or width

`col` Define plotting color
Colors can be defined by index, name, hexadecimal or RGB.
Type `colors()` to see the possible names.

If you want points plotted in different colors, you can create a color column in advance

Matrix of Scatterplots

```
pairs(~mpg+disp+drat+wt,data=mtcars,
      main="Scatterplot Matrix")
```



Histograms and density plots

`hist(x)` creates a histogram, the "`breaks`" option can change the number of bars.

Other graphic parameters (e.g. `main`, `xlab`, `col`) can be used as in "`plot`".

`plot(density(x))` creates a density plot.

Bar plots

`barplot(x)` where x contains the heights of the bars.

If these heights are means of groups in your dataset,

you can calculate them in advance with `aggregate`.

Example:

```
barmeans = aggregate(mtcars$hp, list(mtcars$cyl), mean)
barplot(barmeans$x, main="HP by Cylinders",
        names.arg = levels(mtcars$cyl))
```

Box plots

hp as a function of cyl

```
boxplot(hp~cyl, data = mtcars, main = "HP by Cylinders",
        xlab = "Number of Cylinders", ylab = "HP")
```

The result is similar to the original “plot” function,
but the syntax enables plotting by more than one factor.

```
boxplot(hp~gear*cyl,data = mtcars,
        main = "HP by gears/cylinders",
        xlab = "Number of gears/cylinders", ylab = "HP",
        col = c("green", "yellow", "red"))
```

In this case, we defined 3 colors, but we have 9 boxes.
The sequence of colors is repeated as many times as needed.

Plotting a graph with several panels

```
par(mfrow = c(nrows, ncols))
par(mfcol = c(ncols, nrows))
```

Divides the plotting area into the number of rows and columns you defined. Each new plot you create will be drawn in a new “cell”.

Plotting to a file

You can plot on the screen first and save from the “export” menu,
but you can also plot directly to a file.

You first open the type of file you want with the corresponding function:
`pdf, bmp, jpeg, png, tiff`.
Within each of these functions you can define filename, plot size, etc.

Run all the plotting functions, and close with `dev.off()`

ggplot2

A package that enables creating more complex plots than the basic functions.

Installation: `install.packages("ggplot2")`
This should be done only once per computer, the classroom computers should have this package installed already.

Loading: `library(ggplot2)`
This should be done once per session, to make the commands from `ggplot2` available.

ggplot2 – qplot vs. ggplot

`ggplot` is the main function
`qplot` has limited functionality, but simpler syntax (similar to `plot`)

`qplot` arguments:
`x` – the column to use for the x axis
`y` – the column to use for the y axis
`data` – the data frame containing x, y and other factors if needed
`facets` – split to several plots according to the factor defined here.
`geom` – type of graph (e.g. `"point", "jitter", "line", "bar",` etc.)
`xlim, ylim` – limits for x and y axis
`main, xlab, ylab` – plot title, x axis label, y axis label
`color, size` – change graphic parameters of markers/boxes.

Example:

```
qplot(wt, mpg, data=mtcars, facets=.~gear, color = I("red"),
      size = I(4))
```

This "`I`" is needed to indicate that this is a single value and not a column/factor

ggplot syntax

The logic behind the ggplot syntax is to define the dataset, and then plot by “layers”, connected by “+” signs.
A layer can be scattered markers, bars, error bars, labels, etc.

As an example, we will recreate the same plot we did with `qplot`

```
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point(color="red", size=5) +
  facet_grid(.~gear) +
  xlab("Weight") +
  ylab("Miles per gallon") +
  theme_bw()
```

Of course, there are many more options in ggplot.
Have a look at the ggplot2 cheat sheet in the shared Box folder.

Bar plots with ggplot

The graphic functions for creating a bar plot do not calculate the needed summary statistics (means and s.d. or s.e. for each group), so you will have to do this by yourself.
One option is the function `aggregate`.

```
aggregate(formula = y~x, data = data.frame.name, FUN = mean)
```

column names:
x should be a factor,
y is numeric

any function:
built-in or created by you

In the example, we repeat this several times for different statistics, and join them all to one data frame with `cbind`.

Bar plots with ggplot

```
ggplot(data = averages, aes(x = gear, y = mpg)) +
  geom_bar(stat = "identity", width = 0.5, fill =
           c("red","blue","green"), col = "black") +
  geom_errorbar(aes(ymin = mpg - se, ymax = mpg + se),
                width=0.1) +
  xlab("Number of gears") +
  ylab("Miles per gallon") +
  theme_bw() +
  theme(axis.text.x = element_text(size=16),
        axis.text.y = element_text(size=16),
        axis.title.x = element_text(size=20),
        axis.title.y = element_text(size=20))
```

Practice options

Interactive exercises:

R Swirl:
http://swirlstats.com/

Try R:
http://tryr.codeschool.com/

Look up basic functions:

Quick-R:
http://www.statmethods.net/