Statistical Computing with R

Lecture 6: getting used to R Markdown; lists; data visualization (part 2)

Mirko Signorelli
: mirkosignorelli.github.io
: statcompr [at] gmail.com

Mathematical Institute
Leiden University

Master in Statistics and Data Science (2023-2024)

Recap

Lecture 5:

- loops in `R`
- `for` loops
- special values (`NA`, `NaN`, `Inf` and `NULL`)
- data visualization: part 1
- charts to visualize univariate frequency distributions

Today:

- getting used to `R` Markdown
- lists
- data visualization: part 2
  1. scatter plots
  2. low level graphics functions
  3. functions and curves
  4. exporting images

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

Functions and curves

Exporting images

Preparing for next week's class

Knitting problems

- When creating a document with `R` Markdown, it is important to identify possible compiling errors as soon as possible
- While working on an `.Rmd` file, it is good practice to "knit" (= compile) it regularly
-  Don't wait too long before knitting the file for the first time, and definitely don't do it only at the end! 

Code evaluation: R Markdown vs R scripts / console

Difference between evaluating code in the console and knitting an R Markdown file:

- Every time you knit, R Markdown (re-)evaluates all the R code that it finds in the document (unless `eval = F` or `cache = T`)
- Whatever was executed in the console does not matter for the compilation of the `.Rmd` file! Basically, R Markdown compiles starting from an empty environment

Pro tip: **separate the coding part from the reporting part!**

- If you have a lot of code / time-consuming code, first write your code in an `.R` script. Edit and debut the code there. Only when you are sure that there the code is ok(-ish), copy it into an `.Rmd` file, start structuring the document and focus on the reporting part

Reminder: working directory

- An important difference (already mentioned in Lecture 4, but repetita iuvant):
  1. `R` scripts / console: you need to specify the working directory where to retrieve inputs from and save outputs to (see Lecture 3)
  2. `R` Markdown: by default, the working directory for R code chunks is the directory that contains the Rmd document!
- If you need to change the working directory in `R` Markdown, check out this explanation

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

Functions and curves

Exporting images

Preparing for next week's class

Objects in R

At the beginning of the course, I mentioned that the most common types of objects in R are:

- vectors → L1
- matrices (and arrays) → L2
- data frames (and tibbles) → L2
- lists → today!

Lists

- A list is an R object that contains other objects as elements
- Handy to combine multiple heterogeneous objects into a single one
- Very flexible object type
- In principle, you can nest a list inside another list that is nested inside another list... And so on (⚠ don't try this at home! ⚠)

Creating lists

You can use:

- `list( )` to create a list (unnamed, or named - see examples below)
- `names( )` to assign / retrieve names of objects within the list

```
v = 1:8; df = cars; m = matrix(1:4, 2, 2)
list1 = list(v, df, m) # create an unnamed list
names(list1)
```

```
## NULL
```

```
names(list1) = letters[1:3] # add names
names(list1)
```

```
## [1] "a" "b" "c"
```

```
list2 = list('vector' = v, 'data.frame' = df,
             'matrix' = m) # create a named list
names(list2)
```

```
## [1] "vector"     "data.frame" "matrix"
```

Subsetting lists

- Single elements in lists can be accessed with `[[ ]]`

```
list1[[1]]
```

```
## [1] 1 2 3 4 5 6 7 8
```

- Multiple elements can be subsetted with `[ ]`

```
list1[c(1, 3)]
```

```
## $a
## [1] 1 2 3 4 5 6 7 8
##
## $c
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

[ ] vs [[ ]]

- Difference between [ ] and [[ ]]:

```
list1[1]
```

```
## $a
## [1] 1 2 3 4 5 6 7 8
```

```
list1[[1]]
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
is(list1[1]) # [ ] --> always returns a list!
```

```
## [1] "list"   "vector"
```

```
is(list1[[1]]) # [[ ]] --> returns the actual object :)
```

```
## [1] "integer"                "double"
## [3] "numeric"                "vector"
## [5] "data.frameRowLabels"
```

Subsetting lists (cont'd)

- If named, elements in a list can also be accessed with `$name` or `[['name']]`:

```
list1$a
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
list1[['a']]
```

```
## [1] 1 2 3 4 5 6 7 8
```

Lists (cont'd)

- `length( )` returns the number of objects stored in a list
- An empty list of size k can be created with `vector('list', k)`

```
length(list1)
```

```
## [1] 3
```

```
list3 = vector('list', 4)
length(list3)
```

```
## [1] 4
```

```
list3[[2]]
```

```
## NULL
```

Lists (cont'd)

- c( ) allows to concatenate lists:

```
# how to concatenate 2 lists:
list4 = c(list1, list3)
length(list4)
```

```
## [1] 7
```

```
# how to add an element
# to a list:
list5 = c(list1,
  list('v2' = letters[1:5]))
length(list5)
```

```
## [1] 4
```

Why do we need lists?

Ok Mirko, but...

Why do we even need lists?

Why do we need lists?

Ok Mirko, but...

Why do we even need lists?

- Lists are useful whenever you need / want to bundle a heterogeneous set of objects into a single object
- Particularly useful when you want to return multiple outputs at the end of a function:

```
sample.mean = function(x) {
  n.nas = sum(is.na(x))
  e.x = mean(x, na.rm = T)
  n = length(x) - n.nas
  # create list with all desired outputs
  out = list('mean' = e.x, 'n' = n,
             'n.missing' = n.nas, 'x' = x)
  # return the list as output of the function
  return(out)
}
```

Why do we need lists? (cont'd)

```
v = c(3:5, NA, 13, -4)
sample.mean(v)
```

```
## $mean
## [1] 4.2
##
## $n
## [1] 5
##
## $n.missing
## [1] 1
##
## $x
## [1]  3  4  5 NA 13 -4
```

```
sample.mean(v)$mean
```

```
## [1] 4.2
```

Your turn

Exercises

Camille, Martina and Pedro have so far passed a different number of exams, obtaining the following grades:

- Camilla: 8, 9.5, 9, 8, 7
- Martina: 6.5, 7, 6
- Pedro: 7.5, 8, 9, 8

1. Create a list with each student's grades
2. Compute the mean grade and number of exams passed by each student, and store this information in a data frame

Solutions

```r
# Ex. 1
grades = list(
  'Camilla' = c(8, 9.5, 9, 8, 7),
  'Martina' = c(6.5, 7, 6),
  'Pedro' = c(7.5, 8, 9, 8)
)
grades
```

```
## $Camilla
## [1] 8.0 9.5 9.0 8.0 7.0
##
## $Martina
## [1] 6.5 7.0 6.0
##
## $Pedro
## [1] 7.5 8.0 9.0 8.0
```

Solutions (cont'd)

```
# Ex. 2
df = data.frame(name = names(grades),
                n.exams = NA, mean.grade = NA)
for (i in 1:length(grades)) {
  df$n.exams[i] = length(grades[[i]])
  df$mean.grade[i] = mean(grades[[i]])
}
df
```

```
##      name n.exams mean.grade
## 1 Camilla       5      8.300
## 2 Martina       3      6.500
## 3   Pedro       4      8.125
```

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

Functions and curves

Exporting images

Preparing for next week's class

Data visualization: recap

- Last week we introduced some charts to visualize the distribution of a single variable
- Today we look into ways to visualize the relationship between two variables, ways to change the appereance of a chart created with base `R`, and how to export images you create with `R`

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

Functions and curves

Exporting images

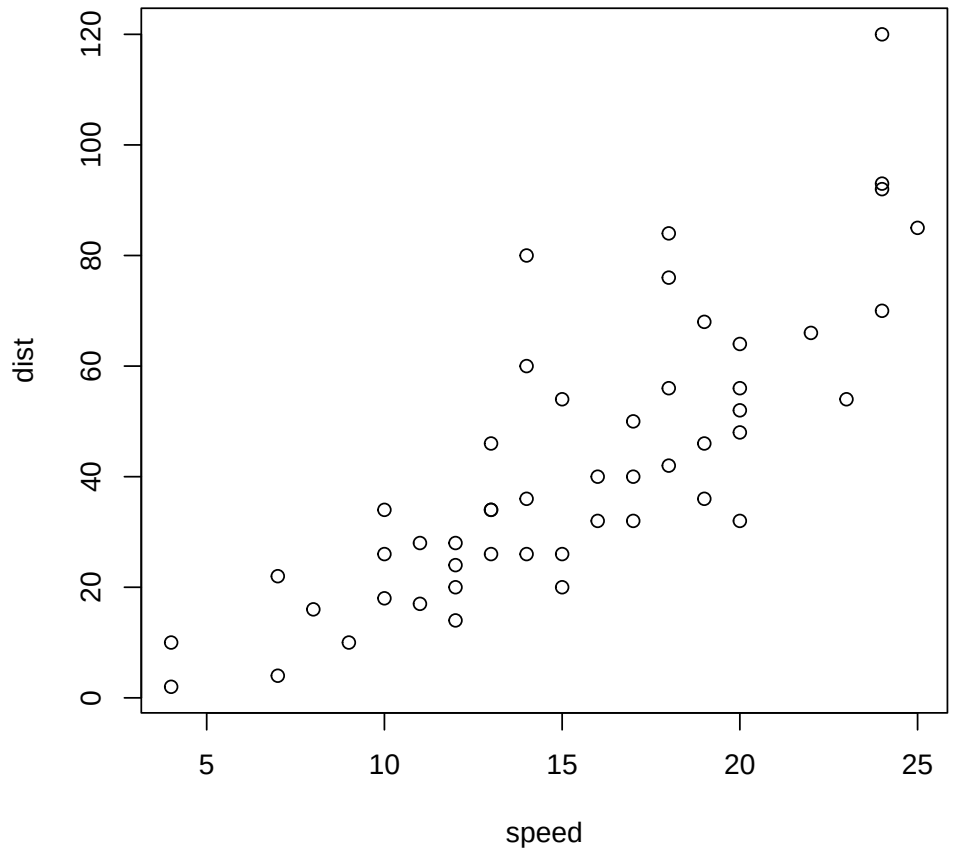
Preparing for next week's class

Scatter plots

- A scatter plot allows to visualize the relationship between two variables
- Base `R` function for scatter plots: `plot( )`
- Several arguments can be used within `plot( )`:
  - `formula` + `data`, or `x` + `y` (see next 2 slides)
  - `main`, `xlab`, `ylab`, `xlim`, `ylim`, `col` like with previous functions
  - `pch`: point type (0, 1, . . . , 25)
  - `type`: p, l, b, c, o, s, h, n
  - `lty`: line type (1 to 6)
  - and many more. . .

Basic scatter plot in R

```
plot(dist ~ speed, data = cars)
```



A bit ugly, isn't it?

Are these expressions equivalent?

- Open R and type:

```
plot(dist ~ speed, data = cars)
plot(cars$speed, cars$dist)
with(cars, plot(speed, dist))
```

- Are these 3 expressions equivalent? Do you notice ANY difference between them?

Some graphics options (`pch`, `type`, `lwd`...)

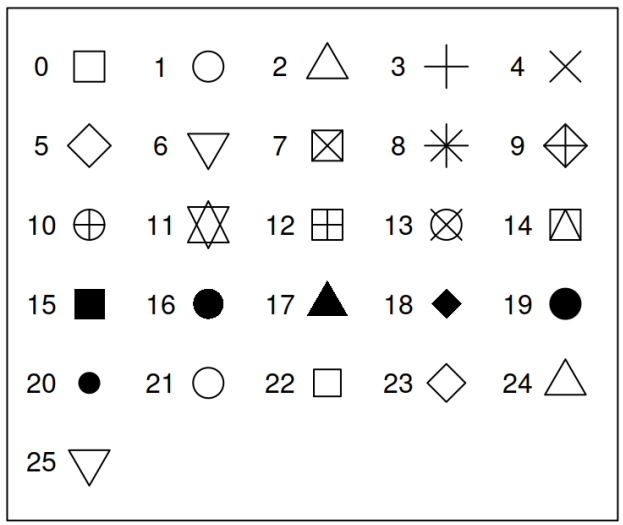


Figure 1: Possible `pch` values

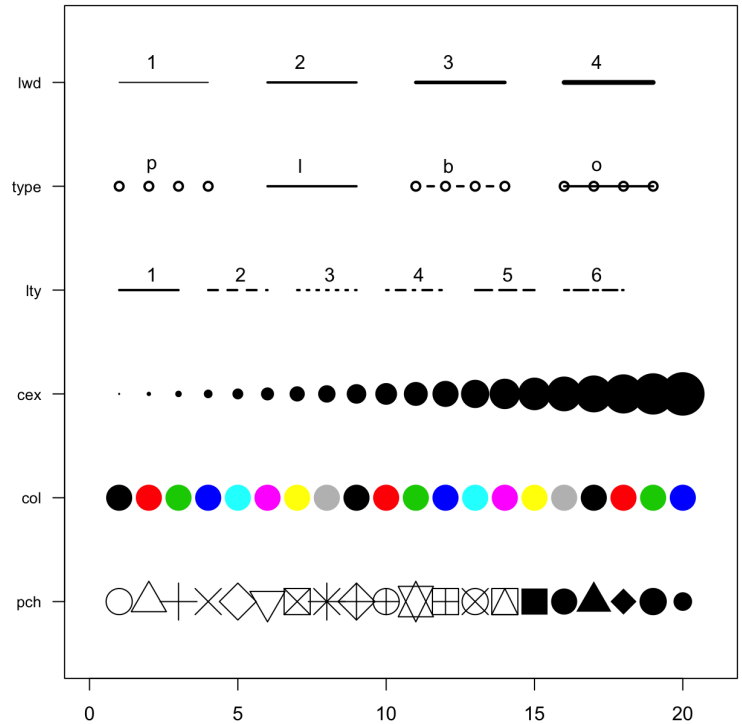
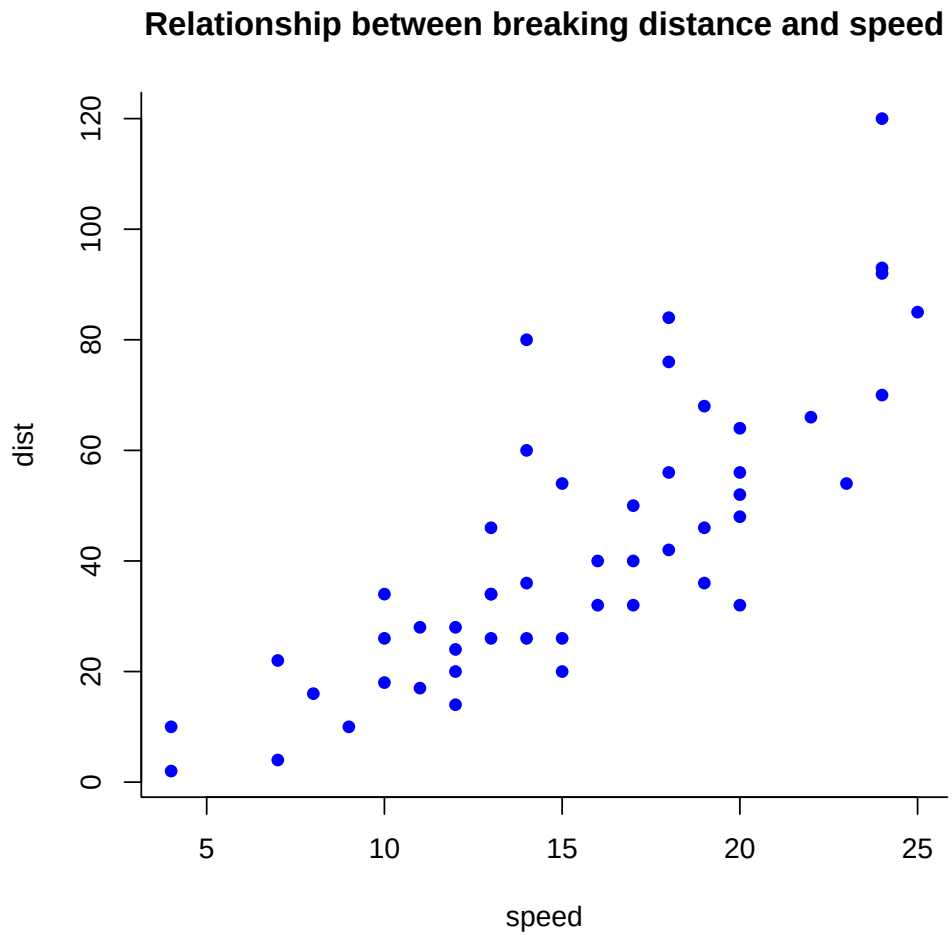


Figure 2: Some graphical parameters that can be supplied to `plot( )`

Making the scatter plot a bit prettier

```
par(bty = 'l')
plot(dist ~ speed, data = cars, col = 'blue', pch = 16,
     main = 'Relationship between breaking distance and speed')
```



Your turn

Exercises

The `iris` dataset contains information on 150 iris plants from different species (see `?iris` for more info about this famous dataset)

1. Make a scatterplot of `Sepal.Width` versus `Sepal.Length`, and color points by `Species`

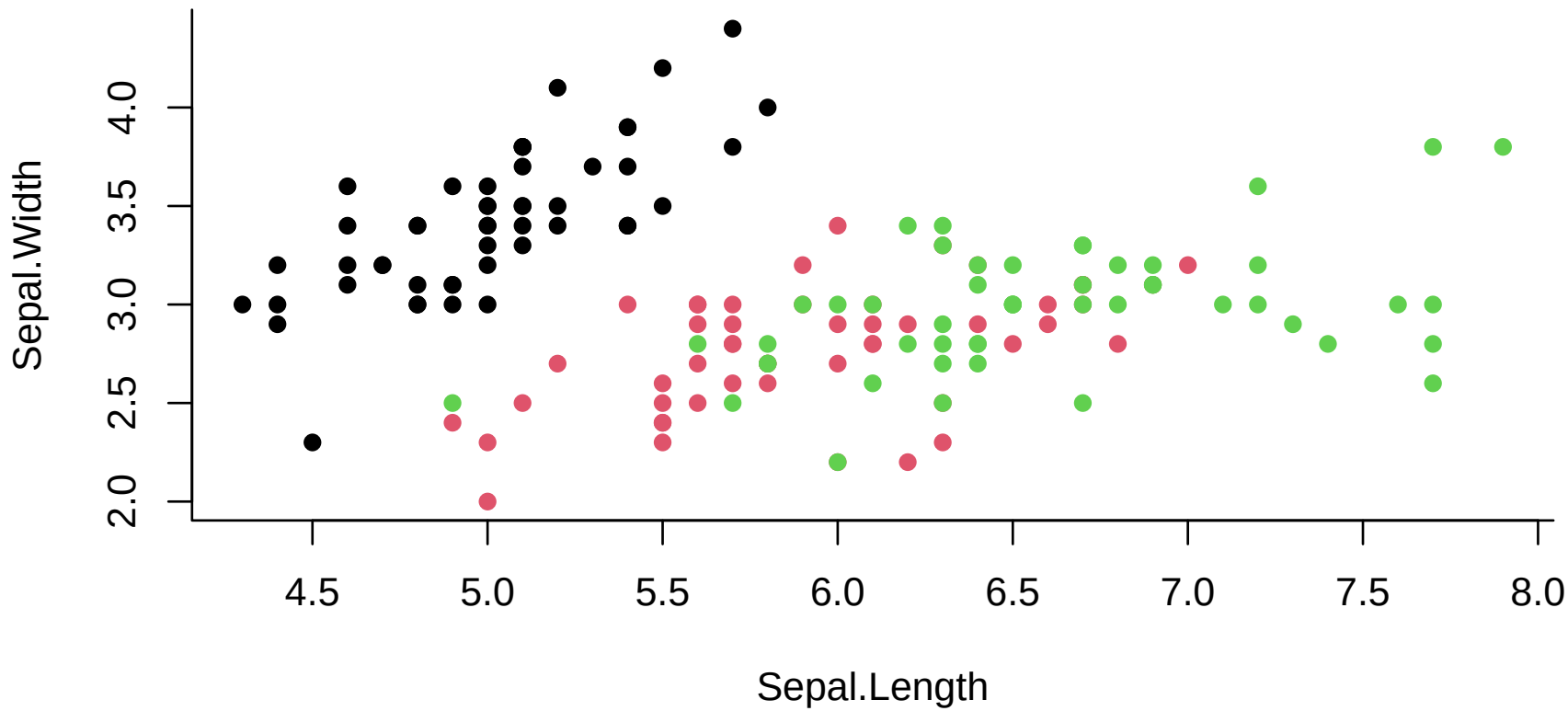
In a few minutes, we will see how we can add a legend to this plot!

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

Solution

```
plot(Sepal.Width ~ Sepal.Length, data = iris,
     col = Species, pch = 16, bty = 'l')
```



Adding elements to a chart

- `plot( )`, `barplot( )`, `pie( )`, `hist( )` are "high level graphics functions" that allow you to quickly create a chart without worrying too much about the details
- They have arguments (`main`, `pch`, ...) that allow you to modify or add some elements of a chart
- How can we add extras (lines, curves, points, legends...) to an existing chart?

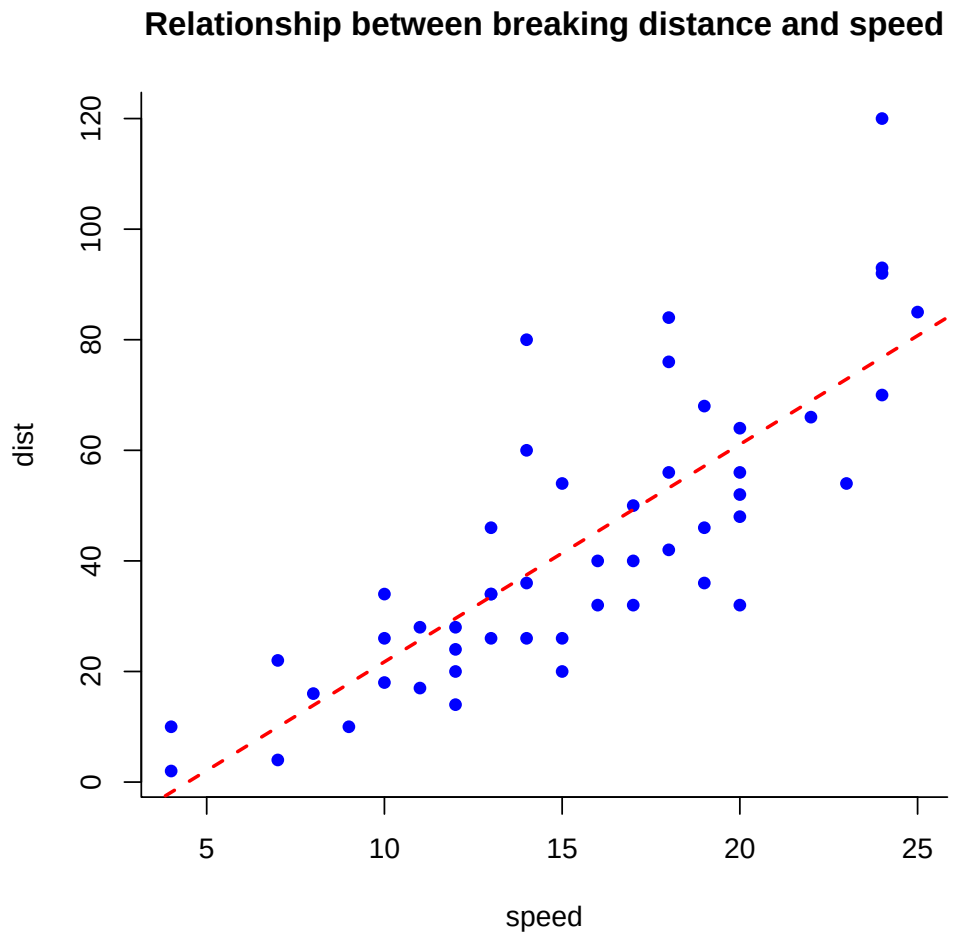
Low level graphics functions

- Further additions to charts from high level graphic functions can be made using low level graphics functions:

```
points(x, y, ...) # adds points
lines(x, y, ...) # adds line segments
text(x, y, labels, ...) # adds text into the graph
abline(a, b, ...) # adds the line $y = a + bx$
abline(h = y, ...) # adds a horizontal line
abline(v = x, ...) # adds a vertical line
polygon(x, y, ...) # draws a (possibly filled) polygon
segments(x0, y0, x1, y1, ...) # draws line segments
arrows(x0, y0, x1, y1, ...) # draws arrows
symbols(x, y, ...) # draws circles, squares, thermometers, etc.
legend(x, y, legend, ...) # draws a legend
```

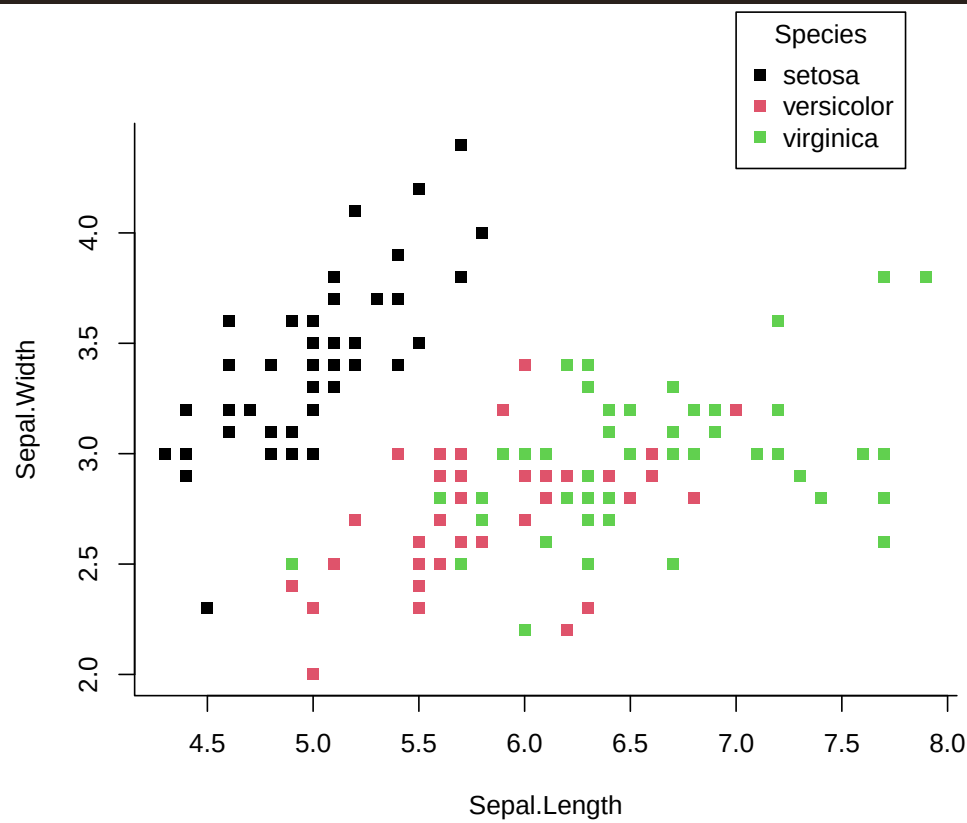
Example: adding the linear regression fit

```
lm.fit = lm(dist ~ speed, data = cars)
b.hat = coef(lm.fit)
abline(a = b.hat[1], b = b.hat[2], col = 'red',
       lty = 2, lwd = 2)
```



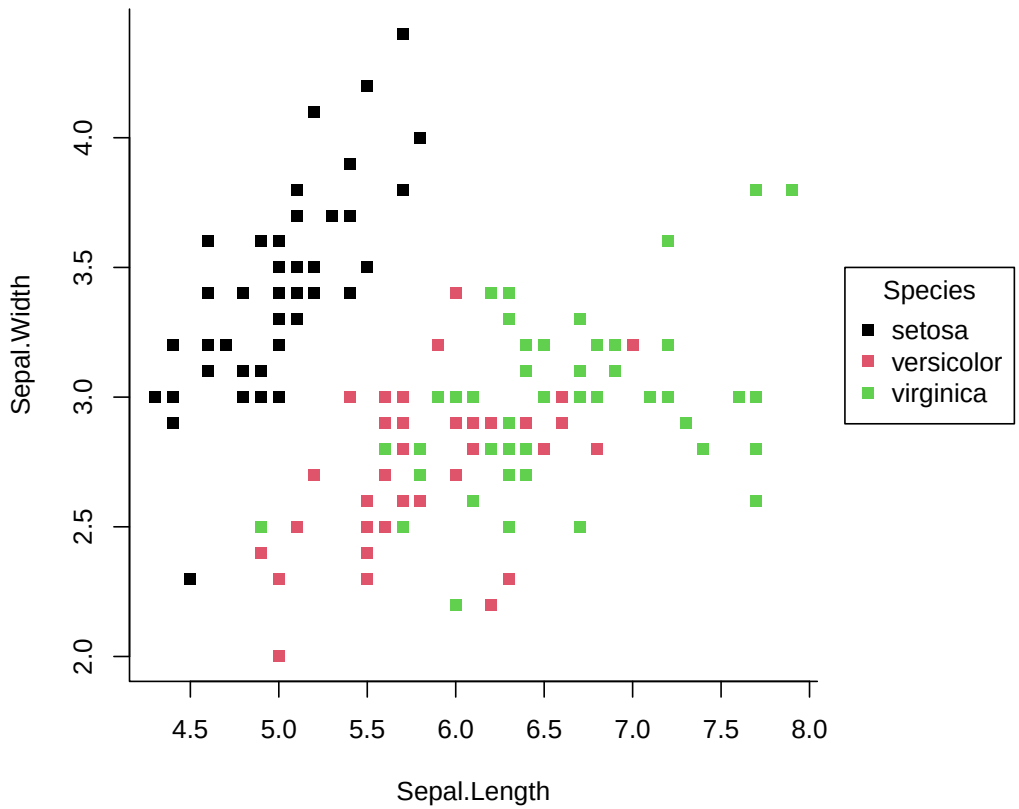
Example: adding a legend (quick and dirty)

```
plot(Sepal.Width ~ Sepal.Length, data = iris,
     col = Species, pch = 15)
legend(7, 5, unique(iris$Species), title = 'Species',
       col = 1:3, pch = 15, xpd = T)
```



Example: adding a legend (with some more effort)

```
par(mar = c(4, 4, 2, 6.5), bty = 'l')
# arguments of par( ) explained in the next slide! :)
plot(Sepal.Width ~ Sepal.Length, data = iris,
     col = Species, pch = 15)
legend(8.2, 3.5, unique(iris$Species), title = 'Species',
       col = 1:3, pch = 15, xpd = T)
```



par( )

The function `par( )` can be used before creating a chart to edit some features of the plot. A selection of arguments:

- `bty` controls the type of border. Possible values: `o, n, l, c, u, 7`. Default is `o`. My favourite? `l` ☺
- `mar`: controls the margins of the plot. Default is `c(5.1, 4.1, 4.1, 2.1)`
- `mfrow`: number of plots within the frame. Default is `c(1,1)`
- `xpd`: is it allowed to draw beyond the plot margins? Default is `FALSE`
- `...` and many more!

⚠ `par( )` changes are semi-permanent. Once an argument is changed, the new argument value continues to apply within the same `R` session until it is changed again! ⚠

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

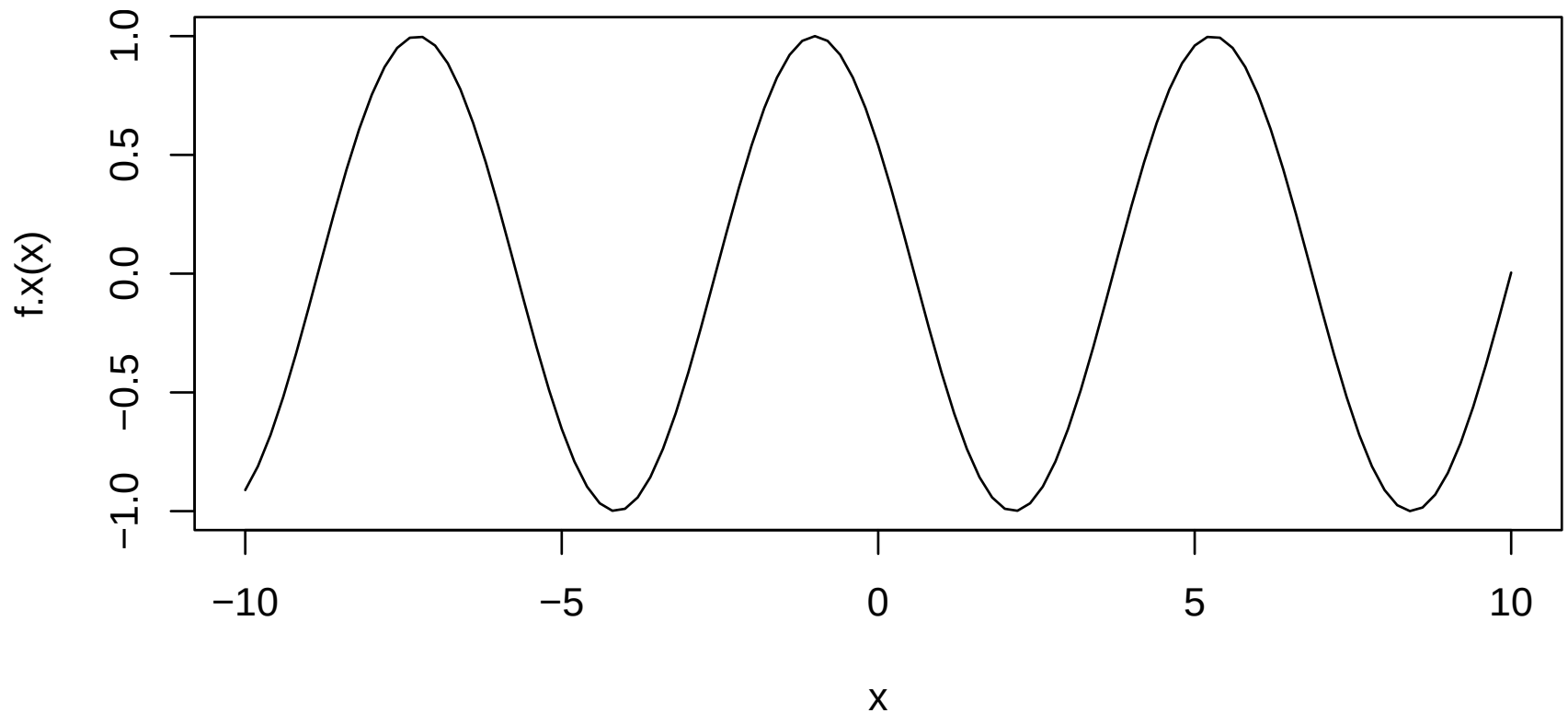
Functions and curves

Exporting images

Preparing for next week's class

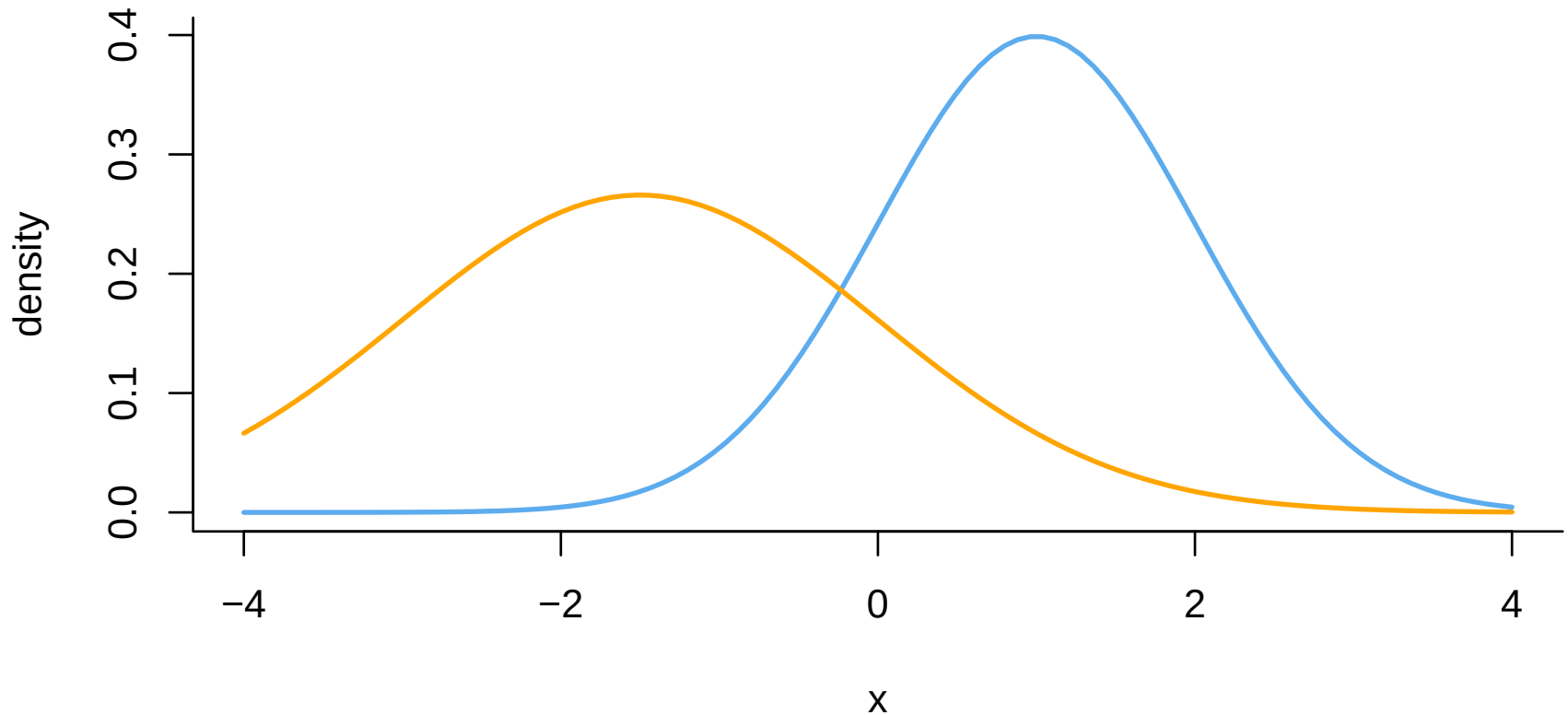
Functions and curves

```r
f.x = function(x) cos(x+1)
curve(f.x, xlim = c(-10, 10))
```



Functions and curves (cont'd)

```
curve(dnorm(x, mean = 1, sd = 1), xlim = c(-4, 4),
      bty = 'l', ylab = 'density',
      col = 'steelblue2', lwd = 2)
curve(dnorm(x, mean = -1.5, sd = 1.5),
      col = 'orange', lwd = 2, add = T)
```



Exporting images

Function	Output type
pdf(...)	PDF
png(...)	PNG
jpeg(...)	JPG
tiff(...)	TIFF

- png, jpeg, tiff: "a single page"
- pdf: you can include multiple pages (see examples in the next slide)

Exporting images (cont'd)

```
# export two charts in a pdf:
pdf('figs/example_pdf.pdf', width = 7, height = 6)
# ?pdf -> width, height are expressed in INCHES!
par(bty = 'l')
with(cars, plot(speed, dist, pch = 15, col = 'blue'))
plot(Sepal.Width ~ Sepal.Length, data = iris,
     col = Species, pch = 15)
dev.off()
# export a chart as jpeg file
jpeg('figs/example_jpg.jpg', width = 600, height = 600)
# ?jpeg -> default for height and width is PIXELS!
par(bty = 'l')
plot(Sepal.Width ~ Sepal.Length, data = iris,
     col = Species, pch = 15)
dev.off()
```

Getting used to R Markdown

Lists

Data visualization (part 2)

Scatter plots

Low level graphics functions

Functions and curves

Exporting images

Preparing for next week's class

About next week's class

- Next week we will wrap up the first half of the course, during which we covered the basics about R
- Homework:
  1. go through the slides of lectures 1-6, and check if there are things that are still unclear
  2. note down any question you may have
  3. I will save some time towards the end of the class to answer your questions