# Statistical Computing with R

## Lecture 8: the `is( )` and `as( )` families; `message( )`, `warning( )` and `stop( )`; `any( )` and `all( )`; data visualization (part 3)

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

Mathematical Institute
Leiden University

Master in Statistics and Data Science (2023-2024)

Universiteit Leiden

# Feedback about assignment 1

- We are currently going through your submitted solutions. Once we finish doing it, we will provide individual feedback through Brightspace
- For now, three general remarks:
  1. pay attention to the formatting of your document. Examples: title should be some sensible text, rather than a file name with _; use headers and subheaders to give structure to your document; avoid that code or output go beyond the page margin...
  2. ex. 1, q5: the dividends in the dataset were paid out quarterly, not yearly!
  3. ex. 1, q7: based on your submissions, we noticed that it makes more sense to store the required information in a data frame, rather than in a list

# Recap

Block 1 (lectures 1-7): R essentials

- object types
- functions and packages
- conditional statements
- loops and `apply( )`
- R Markdown
- dataviz basics
- ... and more!

Block 2 (lectures 8-13): advanced programming topics; probability and statistics with R; more about data visualization

Some of the topics we will cover are:

- warning and error messages
- tracking computing time
- numeric optimization and maximum likelihood
- mixture models and the EM algorithm
- dataviz: `ggplot2`
- `dplyr`; the pipe operator
- horizontal and vertical merges; data in wide and long format

# Today's class

Topics for today:

- the `is` and `as` families
- `message( )`, `warning( )` and `stop( )`
- `any( )` and `all( )` (*self-study*)
- dataviz (part 3): comparing groups and visualizing (many) correlations

# The `is` and `as` families

`message( )`, `warning( )` and `stop( )`

`any( )` and `all( )` (SELF-STUDY)

Dataviz: comparing groups

Dataviz: correlograms

# The `is` family

- When programming, it is often useful to check the type of an object
- This can be done with a set of functions whose name starts with `is`. Examples:

```
is( )
is.data.frame( ) # see also: ?as.data.frame
is.vector( ) # see also: ?as.vector
is.numeric( ) # see also: ?as.numeric
is.character( ) # see also: ?as.character
is.list( ) # see also: ?as.list
is.matrix( ) # see also: ?as.matrix
is.array( ) # see also: ?as.array
is.na( )
```

# Examples

```r
v = 5:9
is(v) # in R, an object can have multiple types
```

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

```r
is.data.frame(v); is.vector(v)
```

```
## [1] FALSE
```

```
## [1] TRUE
```

```r
is.numeric(v); is.character(v)
```

```
## [1] TRUE
```

```
## [1] FALSE
```

Notice the difference:

- `is( )` outputs a vector with all relevant data types
- all other functions output a `TRUE` or a `FALSE`

# Examples (cont'd)

```
v = c(5:9, NA)
is.na(v)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
df = data.frame(letters[1:6], v)
is(df)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

```
is.data.frame(df)
```

```
## [1] TRUE
```

```
is.list(df)
```

```
## [1] TRUE
```

- What the heck is happening here?!?
- Under the hood, ⚠ R stores a data frame as a list of vectors ⚠ where each variable is a vector within the list

# The as family

- Many of the `is` functions have an `as` counterpart
- Members of the `as` family can be used to convert objects from one type to another
- Converting object types can often be useful, but it doesn't always make sense (it depends on what you're trying to do, and what you want to achieve...)

# Example

- What happens if we convert a matrix into a data frame, or into a vector?

```
x1 = matrix(1:8, 2, 4)
x2 = as.data.frame(x1)
x3 = as.vector(x1)
is(x1)
```

```
## [1] "matrix"    "array"     "structure" "vector"
```

```
is(x2)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

```
is(x3)
```

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

# Example (cont'd)

```
x1
```

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

```
x2
```

```
##   V1 V2 V3 V4
## 1  1  3  5  7
## 2  2  4  6  8
```

```
x3
```

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

- Notice how here, `as.vector` has concatenated the columns of `x1` (and not the rows), returning the vector that we supplied to `as.matrix` when creating `x1`

# Example: how to convert characters into dates

- `as.Date( )` can be used to convert `character` dates into dates of type `Date`

```
y1 = '1997-05-27'
y2 = as.Date(y1)
y1; y2
```

```
## [1] "1997-05-27"
```

```
## [1] "1997-05-27"
```

- `y1` and `y2` may look identical, BUT:

```
is(y1); is(y2)
```

```
## [1] "character"           "vector"
## [3] "data.frameRowLabels" "SuperClassMethod"
```

```
## [1] "Date"     "oldClass"
```

# Example: how to convert characters into dates (cont'd)

- If the input date for `as.Date()` is not in the `yyyy-mm-dd` format, you can use the `format` argument to specify the input format

```
as.Date('27-05-1997', format = '%d-%m-%Y')
```

```
## [1] "1997-05-27"
```

```
as.Date('05/27/1997', format = '%m/%d/%Y')
```

```
## [1] "1997-05-27"
```

# Be careful with date formats!

- ⚠ `Y` in `%Y` should be capitalized ⚠

```
as.Date('05/27/1997', format = '%m/%d/%Y') # correct
```

```
## [1] "1997-05-27"
```

- If you use `%y`, things may go wrong:

```
as.Date('05/27/1997', format = '%m/%d/%y') # wrong
```

```
## [1] "2019-05-27"
```

- In this last example with `%y`, `R` takes only the first two digits of 1997 and turns it into 2019. Why?
  - `%y` takes only the first two digits you supplied. If the given number, say `ab`, is between 0 and 68, it turns it into `20ab`. If it's between 69 and 99, it turns it into `19ab`
  - This is extremely confusing $\Rightarrow$ better to avoid `%y`! Always use `%Y` instead

The `is` and `as` families

`message( )`, `warning( )` and `stop( )`

`any( )` and `all( )` (SELF-STUDY)

Dataviz: comparing groups

Dataviz: correlograms

# Motivation

- Many things can go wrong when using functions. Examples:
  - user supplies wrong / unexpected type of input
  - something fails / doesn't work inside the function (examples: matrix not invertible; maximum likelihood estimation failed to converge)
- How can we write functions that inform the user when something is not going as expected?

# `message( )`, `warning( )` and `stop( )`

- `R` has built-in functions that can return different types of messages. Open `R` and type in the console:

```
message("This is a message")
warning("This is a WARNING")
stop("This is an ERROR message")
```

- Notice how `message( )`, `warning( )` and `stop( )` produce different type of messages:

```
> message("This is a message")
This is a message
> warning("This is a WARNING")
Warning: This is a WARNING
> stop("This is an ERROR message")
Error: This is an ERROR message
```

# Example

- Suppose that we want to create a function that computes the median of each column of a matrix (= the equivalent of `colMeans( )`, but for medians)
- Problem: users may supply inputs different from a matrix, for example:
  1. a numeric vector
  2. a character vector
  3. a date
  4. a data frame
  5. a list
- How can we write a function that can deal with the different inputs, and also provide warnings when needed?

# Example (cont'd)

- How to proceed:
  1. think of the possible inputs a user may supply
  2. decide how the function should behave depending on the type of input
- For example, we may want the function to:
  1. output the median of each column if the input is a matrix;
  2. return a warning message if the input is a numeric vector, and then output the median of the supplied vector;
  3. return an error if the input is neither a matrix, nor a numeric vector.
- Let's see how we can implement (1-3)!

# Starting point

1. output the median of each column if the input is a matrix

```
ColMedians = function(x) {
  if (is.matrix(x)) {
    out = apply(x, 2, function(x) median(x, na.rm = T))
  }
  return(out)
}
```

# Step 2: add the warning

2. return a warning message if the input is a numeric vector, and then output the median of the supplied vector

```
ColMedians = function(x) {
  if (is.matrix(x)) {
    out = apply(x, 2, function(x) median(x, na.rm = T))
  }
  else if (is.numeric(x)) {
    warning('Input x should be a matrix, but
             you supplied a vector')
    out = median(x, na.rm = T)
  }
  return(out)
}
```

# `warning( )`

- `warning( )` is used to inform that something unexpected happened, but that computations may nevertheless continue
- When `warning( )` is called:
  - a warning message is generated
  - the evaluation of the function continues
- A `warning( )` is "weaker" / less severe than a `stop( )` (introduced in the next slides)

# Step 3: add the error message

3. return an error if the input is neither a matrix, nor a numeric vector.

```
ColMedians = function(x) {
  if (is.matrix(x)) {
    out = apply(x, 2, function(x) median(x, na.rm = T))
  }
  else if (is.numeric(x)) {
    warning('Input x should be a matrix, but
             you supplied a vector')
    out = median(x, na.rm = T)
  }
  else {
    stop('Input x should be either a matrix
          or a numeric vector')
  }
  return(out)
}
```

# `stop( )`

- `stop( )` is used to inform that something went wrong, and to provide you some hints about what actually went wrong
- When `stop( )` is called:
  - the evaluation of the function is stopped at the line(s) where `stop( )` is evaluated
  - an error message is returned
- `stop( )` is much more severe than `warning( )`

# `message( )`

- Goal of `message( )`: provide useful information (which is neither a warning, or an error message)
  - `message( )` is "weaker" than `warning( )` and `stop( )`
  - Usually less useful and less used
- When `message( )` is called:
  - an informational message is generated
  - the evaluation of the function continues
- See example in the next slide

# Example use of message( )

```
ColMedians = function(x) {
  if (is.matrix(x)) {
    message('Hey, you supplied the right type of input!')
    out = apply(x, 2, function(x) median(x, na.rm = T))
  }
  else if (is.numeric(x)) {
    warning('Input x should be a matrix, but
                you supplied a vector')
    out = median(x, na.rm = T)
  }
  else {
    stop('Input x should be either a matrix
          or a numeric vector')
  }
  return(out)
}
```

NB: this example is only meant to illustrate how `message( )` works. In practice, usually you will not want to include an informational message as silly as the one above in your functions (unless they involve long computations, and you want the user to know at which point the function is!)

# Now let's apply the function!

```
input = cbind(x1 = rbeta(100, 2, 3), x2 = rpois(100, 3))
ColMedians(input)
```

```
## Hey, you supplied the right type of input!
```

```
##        x1        x2
## 0.3849518 3.0000000
```

```
ColMedians(input[ , 1])
```

```
## Warning in ColMedians(input[, 1]): Input x should be a matrix
##                 you supplied a vector
```

```
## [1] 0.3849518
```

- What would happen if you converted `input` into a `data.frame`, and supplied it to `ColMedians`?

# Your turn

## Exercises

Write a function that computes the mean of a vector `x` and behaves as follows:

1. if `x` is not a numeric vector, an error is returned;
2. if `x` is a numeric vector containing `NA`s, a warning is generated, and the mean of `x` (after `NA` removal) is returned;
3. if `x` is a numeric vector without `NA`s, the mean of `x` is returned.

# Solution

```
vec_mean = function(v) {
  err.mess = 'v should be a numeric vector'
  warn.mess = 'v contains NAs'
  if (!is.numeric(v)) stop(err.mess)
  else {
    if (any(is.na(v))) warning(warn.mess)
    mean(v, na.rm = T)
  }
}
```

- any( ) explained in the next slides!

The `is` and `as` families

`message( )`, `warning( )` and `stop( )`

`any( )` and `all( )` (SELF-STUDY)

Dataviz: comparing groups

Dataviz: correlograms

# any( ) and all( )

- In the previous example, we needed to check whether $v$ contained at least one NA

More in general, we may want to check:

- if a condition is satisfied at least once → any( )
- if a condition is always satisfied → all( )

```
v1 = c(T, F, T)
any(v1)
```

```
## [1] TRUE
```

```
all(v1)
```

```
## [1] FALSE
```

# Examples

```
v2 = c(1, NA, 3)
all(is.na(v2))
```

```
## [1] FALSE
```

```
any(is.na(v2))
```

```
## [1] TRUE
```

```
v3 = c(NA, NA, NA)
any(is.na(v3))
```

```
## [1] TRUE
```

```
all(is.na(v3))
```

```
## [1] TRUE
```

# Bonus question

- What if I need to check that exactly / more than $k$ conditions are true?
- Solution: use `sum( )`!

```
v4 = 1:5
# condition: each value is > 2
v4 > 2
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE
```

```
# are exactly two conditions true?
sum(v4 > 2) == 2
```

```
## [1] FALSE
```

```
# are there at least two conditions satisfied?
sum(v4 > 2) >= 2
```

```
## [1] TRUE
```

The `is` and `as` families

`message( )`, `warning( )` and `stop( )`

`any( )` and `all( )` (SELF-STUDY)

Dataviz: comparing groups

Dataviz: correlograms

# Comparing groups

Problem: we want to draw a chart (a single one!) to compare

- the distribution of a quantitative variable across different groups
- the distribution of different quantitative variables

Most common options:

1. density plot
2. boxplot
3. violin plot

# split( )

- `split( )` allows to split a data frame into multiple data frames, each corresponding to a different group
- Output: a list of data frames!

```
iris.split = split(iris, iris$Species)
is(iris.split)
```

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

```
names(iris.split)
```

```
## [1] "setosa"     "versicolor" "virginica"
```

```
is(iris.split)
```

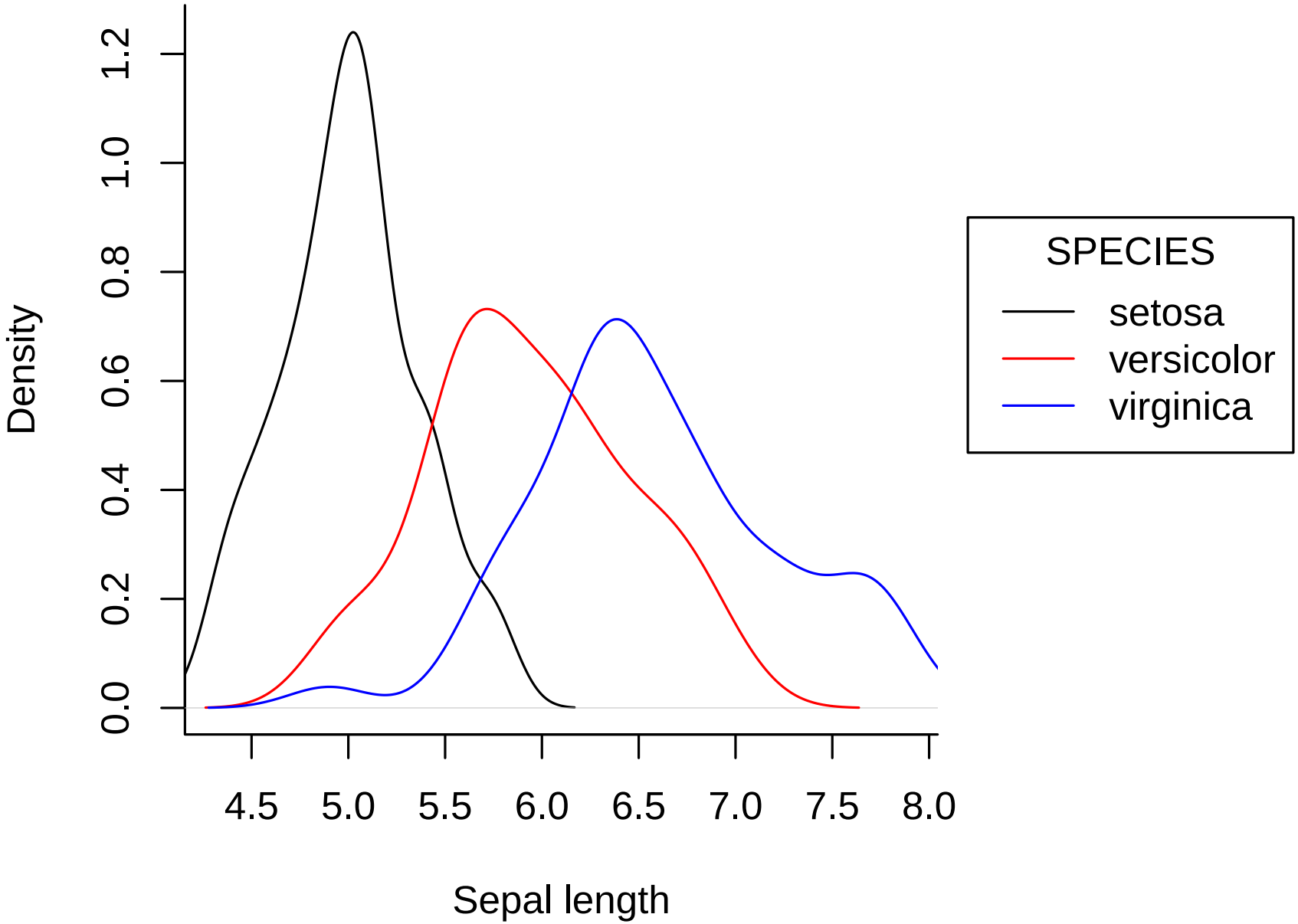
```
## [1] "list"   "vector"
```

# Density plot

- We already considered the density plot as a way to visualize a continuous distribution in L5
- The same type of plot can be used to compare different densities:

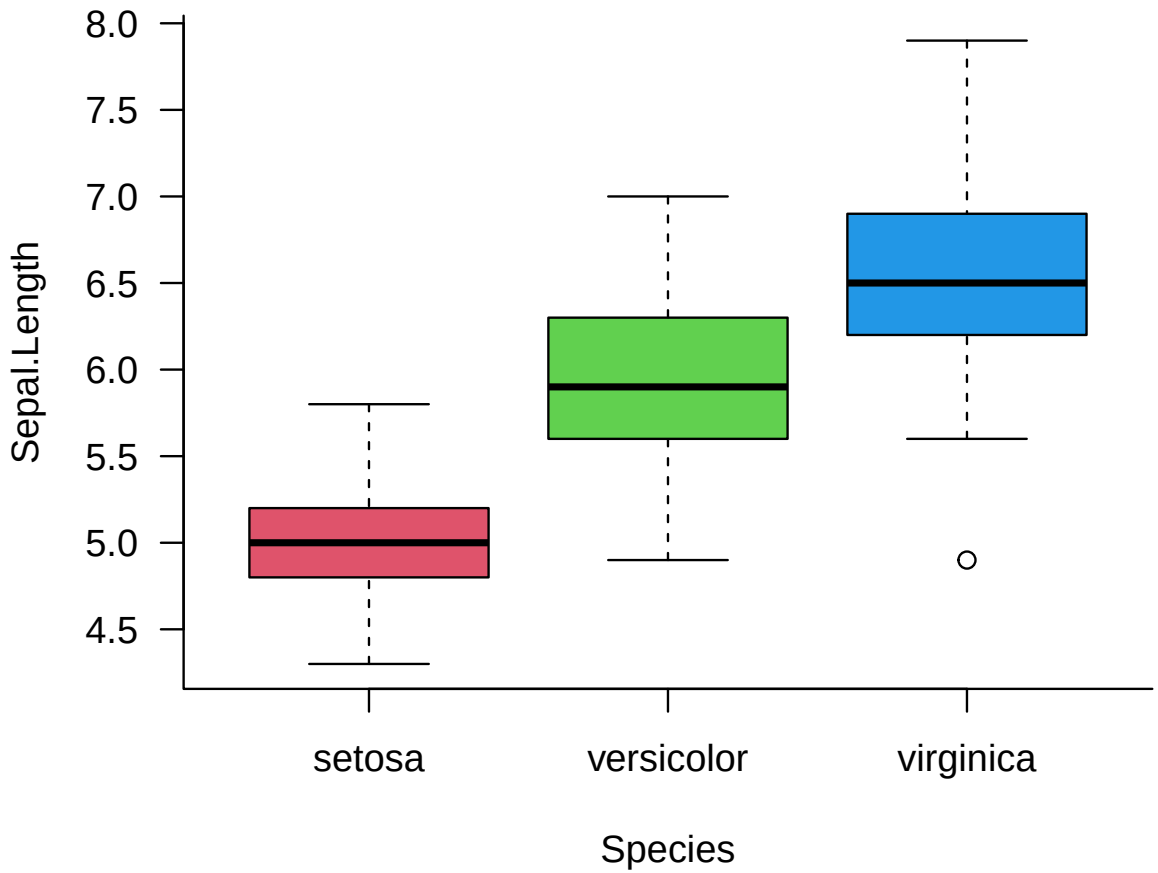
```
iris.split = split(iris, iris$Species)
d.set = density(iris.split$setosa$Sepal.Length)
d.vers = density(iris.split$versicolor$Sepal.Length)
d.virg = density(iris.split$virginica$Sepal.Length)

par(mar = c(5, 4, 2, 10), bty = 'l')
plot(d.set, main = '', xlab = 'Sepal length',
     xlim = range(iris$Sepal.Length))
lines(d.vers, col = 'red')
lines(d.virg, col = 'blue')
legend(8.2, 0.9, legend = unique(iris$Species),
       title = 'SPECIES', lwd = 1, xpd = T,
       col = c('black', 'red', 'blue'))
```

# Density plot (cont'd)



# Boxplot

```
par(bty = 'l')
boxplot(Sepal.Length ~ Species, data = iris,
        col = 2:4)
```



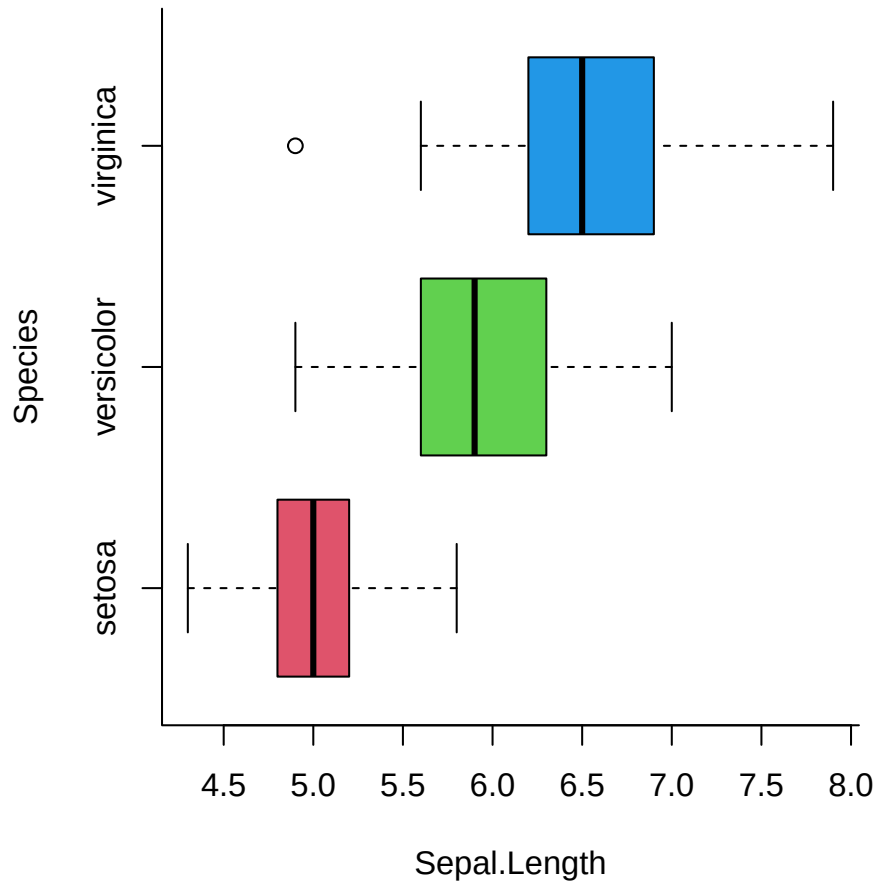
# What do the elements in a boxplot mean?

What do the elements in a boxplot correspond to?

- Box $= 1^{st}$ and $3^{rd}$ quartile (Q1 and Q3)
- Center of the box $=$ median (Q2)
- Whiskers:
  - smallest observed value $> Q1 - 1.5(Q3 - Q1)$
  - largest observed value $< Q3 + 1.5(Q3 - Q1)$

# Horizontal boxplot

```
par(bty = 'l')
boxplot(Sepal.Length ~ Species, data = iris,
        col = 2:4, horizontal = T)
```

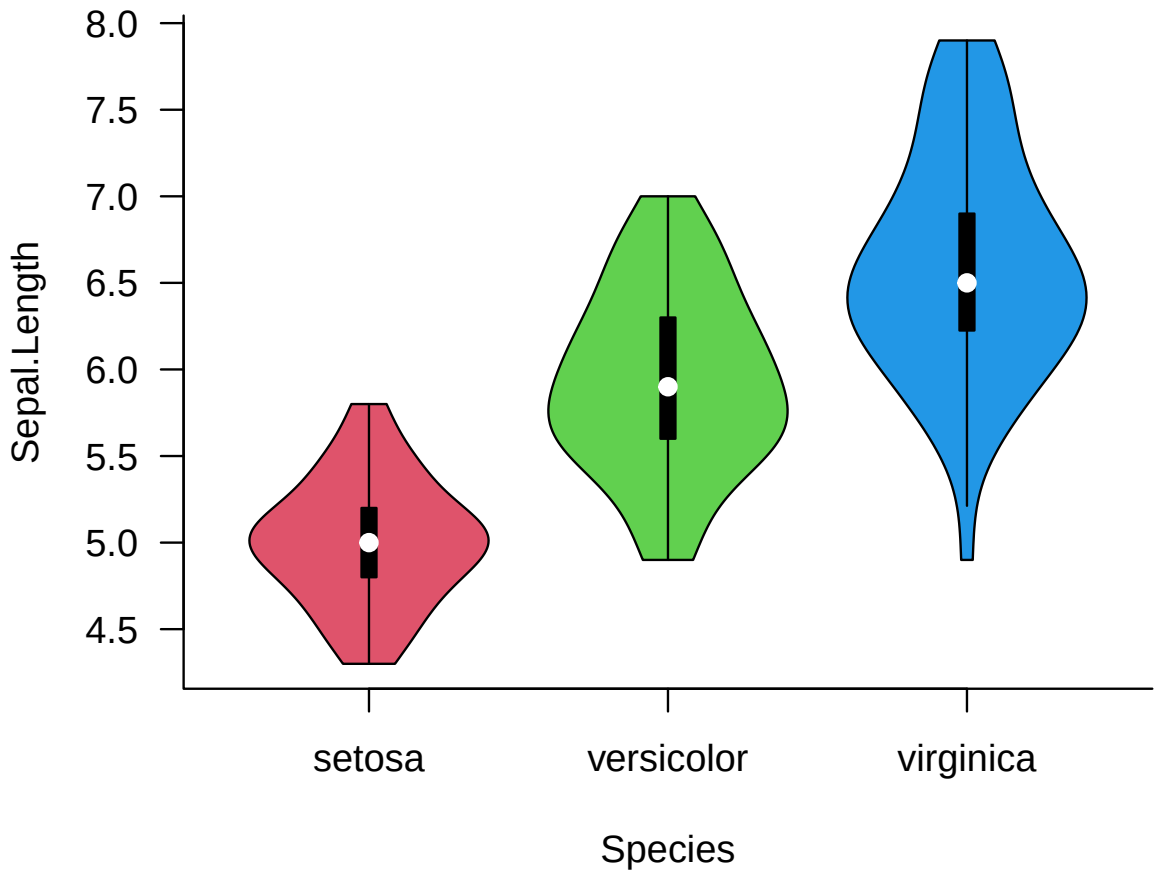


# Violin plot

- Density plots display the whole distribution, and they may make comparisons difficult
- Boxplots display summary statistics, making comparisons easier
- Violin plots combine boxplots and density plots
  - "Inside": a boxplot
  - "Outside": a coloured area that is proportional to the density of your variable
  - Result: a plot that looks like a violin (hence the name)
  - See next slide for an example!

# Violin plot (cont'd)

```r
# install.packages("vioplot")
library(vioplot)
par(bty = 'l', las = 1)
vioplot(Sepal.Length ~ Species, data = iris, col = 2:4)
```



The `is` and `as` families

`message( )`, `warning( )` and `stop( )`

`any( )` and `all( )` (SELF-STUDY)

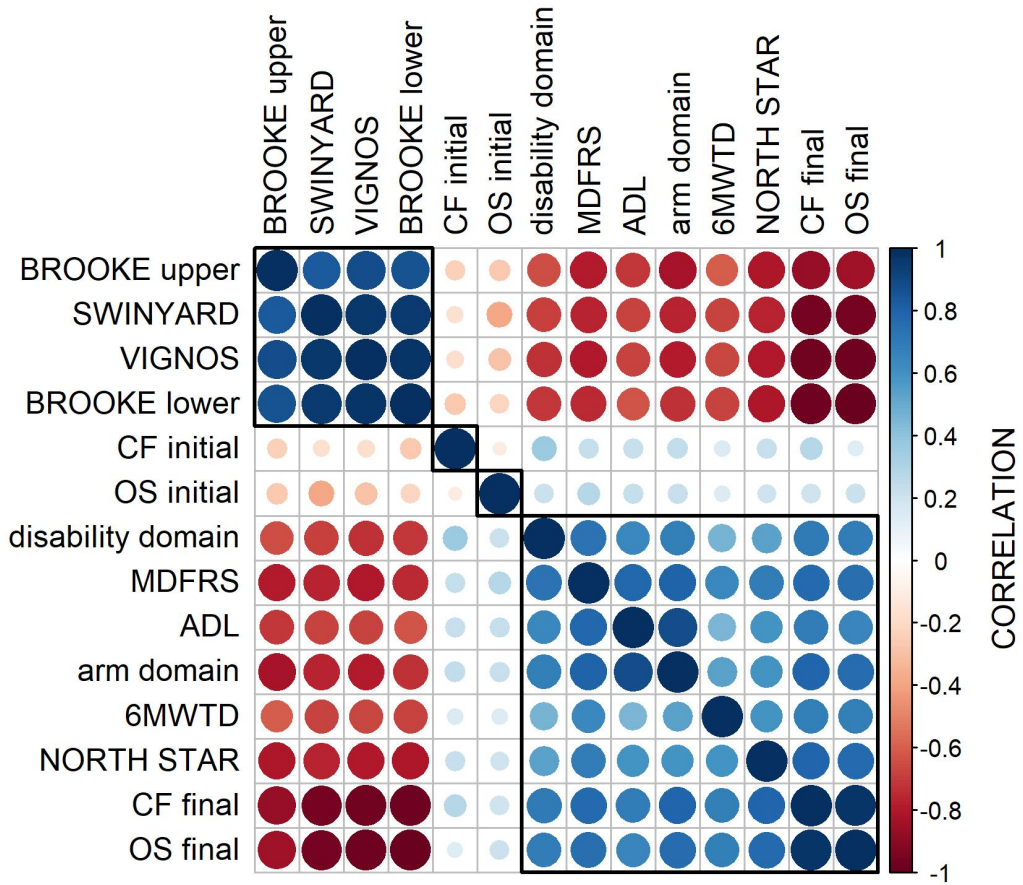
Dataviz: comparing groups

Dataviz: correlograms

# Visualizing correlations

- Problem: suppose that you have a data frame with many quantitative variables. How can you visualize them all in a plot?
- Easy solution: a correlogram (= a graph that visualizes correlations)

# Example of a correlogram




Example taken from: Signorelli, Ebrahimpoor et al. (2021). Peripheral blood transcriptome profiling enables monitoring disease progression in dystrophic mice and patients. *EMBO Molecular Medicine*, 13, e13328 (supplementary figure 6)

# Correlogram

```
#install.packages("corrplot")
library(corrplot)
```

```
## corrplot 0.92 loaded
```

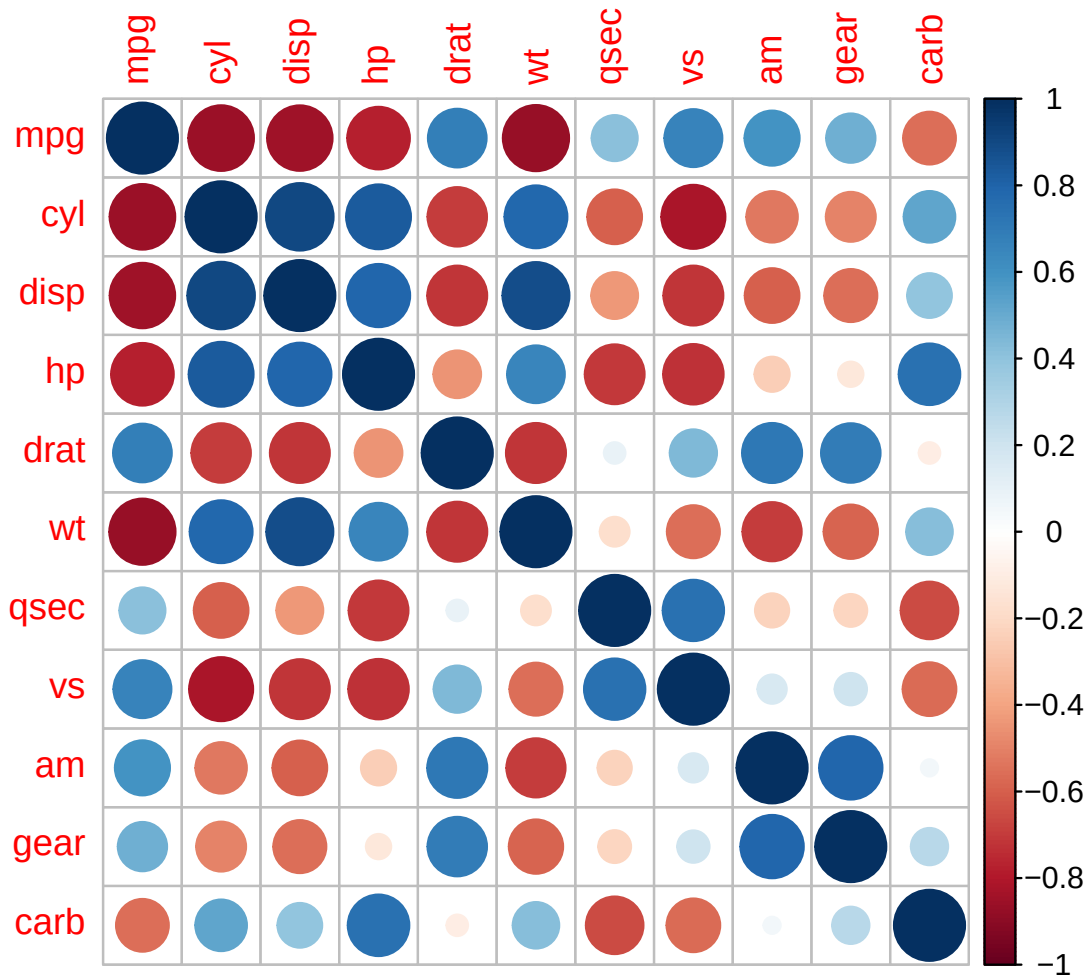
```
head(mtcars, 4)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs
## Mazda RX4        21.0   6  160 110 3.90 2.620 16.46  0
## Mazda RX4 Wag    21.0   6  160 110 3.90 2.875 17.02  0
## Datsun 710       22.8   4  108  93 3.85 2.320 18.61  1
## Hornet 4 Drive   21.4   6  258 110 3.08 3.215 19.44  1
##                  am gear carb
## Mazda RX4         1    4    4
## Mazda RX4 Wag     1    4    4
## Datsun 710        1    4    1
## Hornet 4 Drive    0    3    1
```

```
# step 1: compute correlations
corr.matr = cor(mtcars)
```

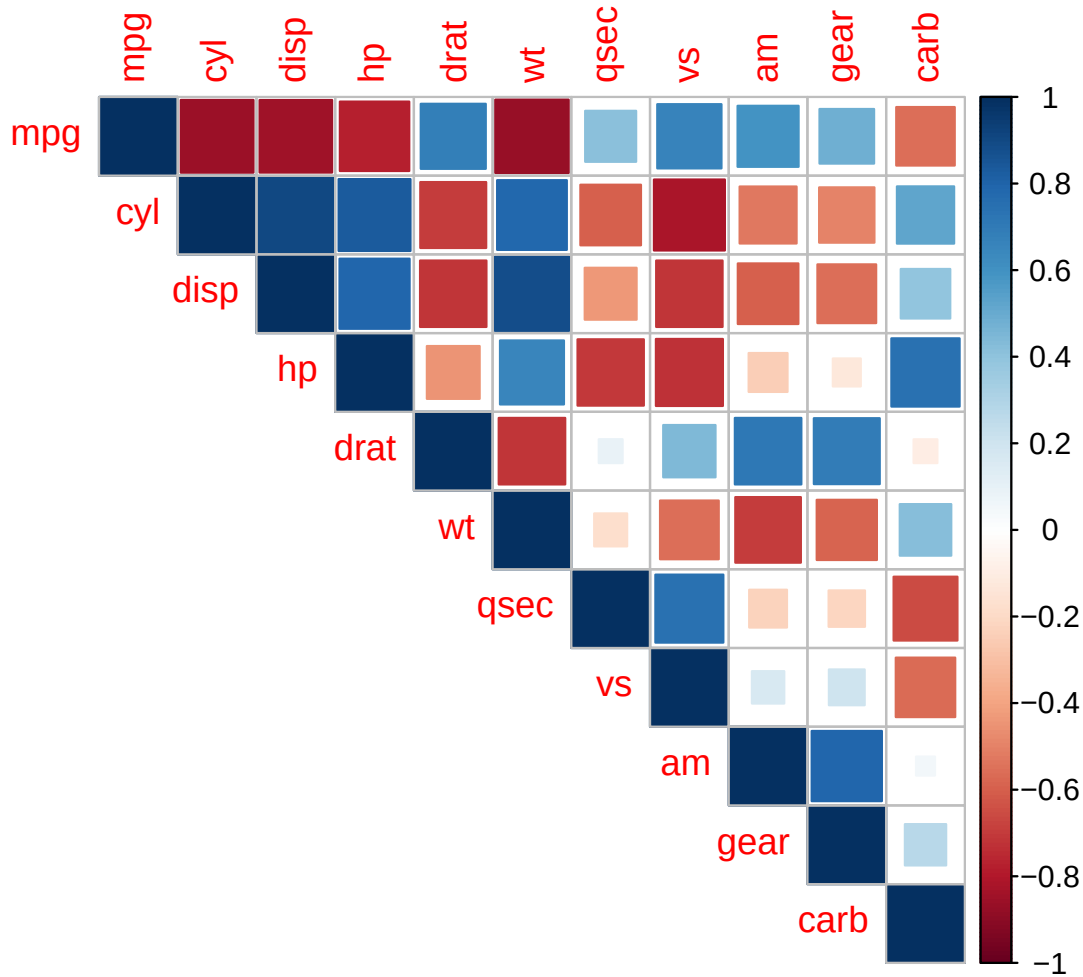
# Correlogram (cont'd)

```
corrplot(corr.matr, method="circle")
```



# Correlogram (cont'd)

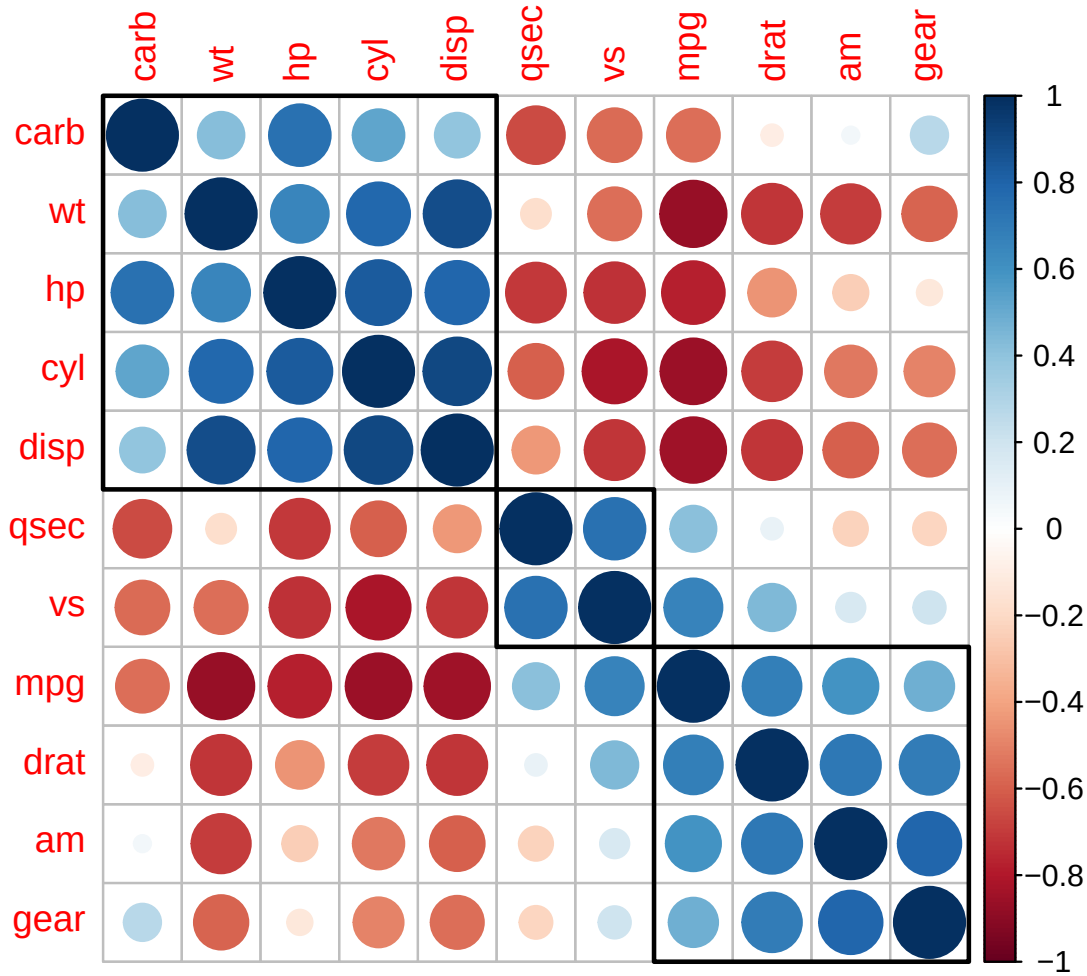
```
corrplot(corr.matr, method="square", type="upper")
```



# How to cluster the correlations?

- See `?corrplot` for the several options!

```
corrplot(corr.matr, method="circle", order = 'hclust',
         hclust.method = 'complete', addrect = 3)
```



# Correlogram (cont'd)

More about correlograms:

- vignettes of the `corrplot` package
- STHDA blog post