University of Zurich[UZH]

URPP Evolution in Action

URPP tutorial

R basic tutorial I

Dr. Heidi E.L. Tschanz-Lischer
University of Zurich
Switzerland

20 November, 2017

Outline

1. Repeating things:
   Introduction to ways of automating repetitive tasks
2. Control structures
3. Missing data
4. Bad habits in coding
5. R graphics

Test data set

- We will work with a test data set of R:
  - To see all test data sets of R: `data()`
  - Iris dataset (Edgar Anderson's Iris Data): `data(iris)`
  - Summary of the dataset: `summary(iris)`

```
summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa     :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Quantify the morphologic variation of three *Iris* flowers:

- setosa
- versicolor
- virginica

Sepal and petal length/width

Ways of automating repetitive tasks

- Many times you have to repeat things

- Common problem:
  - copy-and-pasted some piece of code a bunch of times
  - changing one thing each time

  → Script gets very long
  → Hard to read
  → Bugs accumulate

Apply a function over and over again

- Use the iris data set:

```
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

- We want to get the mean of each column:

```
mean(iris$Sepal.Length)
mean(iris$Sepal.Width)
mean(iris$Petal.Length)
mean(iris$Petal.Width)
```

→ Very repetitive!

sapply

- We can use **`sapply`**: apply a Function over a List or Vector:
  - first argument: a list (or vector)
  - applies a function to each element

- Example:

```
obj <- list(a=1:5, b=c(1, 5, 6))
obj
$a
[1] 1 2 3 4 5

$b
[1] 1 5 6

sapply(obj, length)
a b
5 3
```

← computes the length of each element in `obj`
(this works for any function!)

sapply

Apply it to estimate means of each column in the iris data set. Additionally estimate variance and standard deviation.
(hint: use `iris[,1:4]` to select first 4 columns)

```
mean(iris$Sepal.Length)
mean(iris$Sepal.Width)
mean(iris$Petal.Length)
mean(iris$Petal.Width)
```

sapply

- `sapply` can also be used with an own function:

```
range <- function(x){
  max(x)-min(x)
}
sapply(iris[,1:4],range)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
         3.6          2.4          5.9          2.4
```

- Also possible to supply more than one argument:

```
getQuantile <- function(x,probability){
  quantile(x,probs=probability)
}
sapply(iris[,1:4],getQuantile,0.3)
Sepal.Length.30%  Sepal.Width.30% Petal.Length.30%  Petal.Width.30%
            5.27             2.80             1.70             0.40
```

Additional arguments are put behind function name

lapply

- **sapply**: - apply a Function over a List or Vector
  - returns a vector or matrix
- **lapply**: - apply a Function over a List or Vector
  - returns a list

```
obj <- list(a=1:5, b=c(1, 5, 6))
sapply(obj, length)
a b
5 3

lapply(obj, length)
$a
[1] 5

$b
[1] 3
```

apply

- To apply a function over an array or matrix use **`apply`**:
  - Second argument:
    - 1: apply function over rows
    - 2: apply function over columns
  - Third argument: function to apply

```
matrix.irisSe <- as.matrix(iris[iris$Species=="setosa",1:4])
apply(matrix.irisSe[,1:4],1,mean)
  1     2     3     4     5     6     7     8     9    10    11
2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400 2.700 …

apply(matrix.irisSe[,1:4],2,mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.006        3.428        1.462        0.246
```

Apply a function over each treatment

- Often one wants to apply a function over each treatment/sample
- Example: mean sepal length for each iris species:

```
mean(iris$Sepal.Length[iris$Species=="setosa"])
mean(iris$Sepal.Length[iris$Species=="versicolor"])
mean(iris$Sepal.Length[iris$Species=="virginica"])
```

- **tapply**:
  - apply a Function over a List or Vector
  - Second argument: is the grouping variable (list of one or more factor)
  - Third argument: function to apply

```
tapply(iris$Sepal.Length,iris$Species,mean)
 setosa versicolor  virginica
      5.006      5.936      6.588
```

Summary: apply functions

function	apply to	returns	special
sapply	- List - vector	- Vector - matrix	
lapply	- List - vector	- List	
apply	- Array - Matrix	- Vector - Array - List	
tapply	- Vector	- Array	Apply function to each factor separately

loops

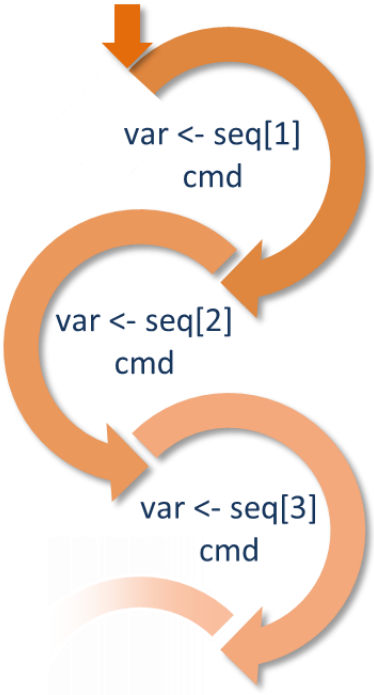
You can also use loops to repeat things

- for: If you know in advance the values that the loop variable will have each time it goes through the loop
- while: If you do not know in advance how often the instructions will be executed

For-loop:

- controlled by a looping vector
  → do something for all items in the vector
- In every iteration of the loop one value in the looping vector is assigned to a variable

```
for(var in seq){
   cmd
}
```



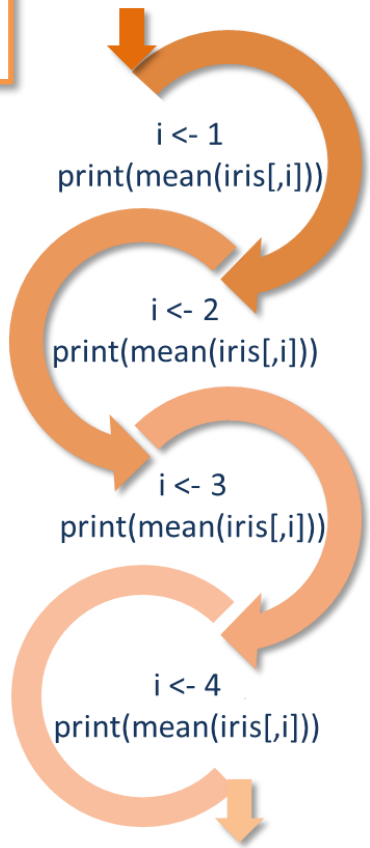
For-loop

- Everything we did before can also be done with a for-loop:

```
sapply(iris[,1:4],mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

```
for(i in 1:4){
  print(mean(iris[,i]))
}
[1] 5.843333
[1] 3.057333
[1] 3.758
[1] 1.199333
```

In a loop `print` is needed to make results visible



For-loop

- You can also store results of a for-loop in a vector:

```
out <- numeric(4)
for(i in 1:4){
  out[i] <- mean(iris[,i])
}
out
[1] 5.843333 3.057333 3.758000 1.199333

names(out) <- names(iris[,1:4])
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

Add names to the `out` vector

For-loop

- There are times where loops are absolutely the best (or the only way) of doing something
- Example: if one wants to get cumulative sum over the elements

```
out <- numeric(4)
total <- 0       # running total
for(i in 1:4) {
  total <- total + sum(iris[,i])
  out[i] <- total
}
names(out) <- names(iris[,1:4])
out
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       876.5       1335.1       1898.8       2078.7

sapply(iris[,1:4],sum)
 Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       876.5       458.6        563.7        179.9
```

Apply/for-loop exercises

Use the data set Orange (`data(Orange)`):

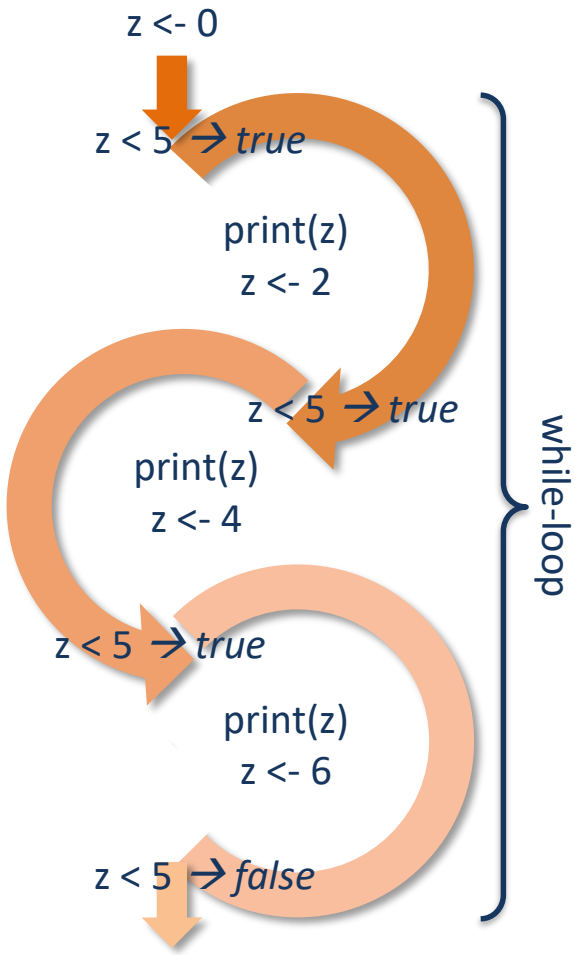
- Estimate mean and sd of the age and circumference across all trees
- Estimate mean and sd of circumference for each tree separately
- Use for-loop to estimate the minimum and maximum of the age and circumference across all trees

while loop

- controlled by a conditional statement
  → do something while condition is true

```
while(cond1=true){
   cmd
}
```

```
z <- 0
while(z < 5) {
   print(z)
   z <- z + 2
}
[1] 0
[1] 2
[1] 4
```



Control structures

- Control structures:
  a block of programming that analyzes variables and chooses a direction in which to go based on given parameters
- basic decision-making process: choose' which lines of code to execute depending on the state of the variables inside the control structure
  - could make the code jump to a completely different part of the program
  - could make it re-run a certain piece again
  - just skip a bunch of code
- immensely important for programming
  → they make the programs function properly
- without control structures the program would only flow in one way
  → Changing what the code does based on a variable is what makes programs useful!

If-else statement

- If-else statements:
  can be used to check for something and do something else depending on the outcome of the check
- if-else can be combined unlimited

```
if(cond1=true) {
   cmd1
} else if(cond2=true){
   cmd2
} else {
   cmd3
}
```

→ Avoid inserting newlines between '} else'

```
number <- 0
if(number < 0){
   print("negative")
} else if (number == 0){
   print("zero")
} else {
   print("positive")
}
[1] "zero"
```

Conditional statements

Comparison operators:

Operator	Meaning
==	Equality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Logical operators:

Operator	Meaning
&&	AND (both statements true)
||	OR (at least one statement true)
!	NOT

```
if(a < 4 && b < 4){
 print("both < 4")
} else {
 print("min one >= 4")
}
```

```
if(a < 4 || b < 4){
 print("min one < 4")
} else {
 print("both >= 4")
}
```

```
if(a != 4){
 print("a not 4")
} else {
 print("a is 4")
}
```

Control structures exercises

while-loop:

Write some lines of code that will figure out
how many terms in the sum 1+2+3+...
it requires for the sum to exceed one million

→ Hint: Create a variable that will store the current sum and another variable that keeps track of what number you are adding to the sum

Condition:

Write a short program that lists
all numbers from 1 to 100
that are divisible by 2 and 3

→ Hint: use 5%%2 to get the rest of a division = 1

Missing data

- Missing data are quite common in biology
  (Individuals die, equipment breaks, you forget to measure something,…)
- If you load in data with blank cells, they will appear as: `NA`
- `NA` is not a string (`"NA"`)! → number that could stand in for anything
- Artificially insert an missing value in our iris data set:

```
iris[2,1] <- NA
head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2           NA         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Missing data

- If we estimate the mean of something which contains missing values, the result will also be missing
  (mean of anything will be anything)

```
iris[2,1] <- NA
mean(iris[,1])
[1] NA
```

- A lot of functions therefore include an option to remove missing values:

```
mean(iris[,1],na.rm=TRUE)
[1] 5.849664

sum(iris[,1],na.rm=TRUE)
[1] 871.6
```

Missing data

- To check for missing data: `is.na()`
- To remove missing data from the data set: `na.omit()`

```
iris[2,1] <- NA
is.na(iris[2,])
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
2         TRUE        FALSE        FALSE       FALSE   FALSE

iris.na <- na.omit(iris)
head(iris.na)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
7          4.6         3.4          1.4         0.3  setosa
```

→ removes whole second row

Missing data – Exercise

- This standard error function doesn't deal with missing values:

```
standard.error <- function(x) {
  v <- var(x)
  n <- length(x)
  sqrt(v/n)
}
standard.error(iris$Sepal.Length)
[1] NA
```

Rewrite it that it filters missing data

Bad habits in coding

1. Avoid indexing by location:

```
#avoid:
iris[,1]
#use instead:
iris$Sepal.Length
```

→ If you change the order, add or delete a columns, the index may change

→ More robust and easier to read

2. Don't use `T` and `F` as shortcuts for `TRUE` and `FALSE`

→R allows to overwrite `T` but not `TRUE`

```
T <- "hello"
T
[1] "hello"
TRUE <- "hello"
Error in TRUE <- "hello" : invalid (do_set) left-hand side
to assignment
```

Bad habits in coding

3. Don’t be inconsistent with assignment symbols, variable names,…
   → easier to revisit the code in the future
4. Avoid copy and pasting large chunks of code between projects
   → Make functions and copy those
5. Don’t write enormously long functions
   → difficult to read
   → each function should do just one thing
6. Don’t write functions that depend on global variables
   → hard to use those functions elsewhere
   → All data should be given as arguments to the function

Bad habits in coding

7. Document your work
   - → You probably cannot have enough describing where data files came from, what scripts do what, how particular pieces of code work
   - → Avoid writing comments that just repeat what the code says
   - → Write comments that describe your intent, reasons for approaches, sources of data / code / algorithm
   - → Will make your life easier when you come back to a project

8. Keep things cleaned up
   - → Tidy computing workspaces are as important as tidy lab workspaces

Bad habits in coding

9. Indent your code and do it consistently
   - → Makes the difference between easy and hard to read code
   - → Indent depth implies something about program structure

```
standard.error <- function(x) {
  v <- var(x)
n <- length(x)
sqrt(v/n)
}
standard.error(iris$Sepal.Length)
```

```
standard.error <- function(x) {
  v <- var(x)
  n <- length(x)
  sqrt(v/n)
}
standard.error(iris$Sepal.Length)
```

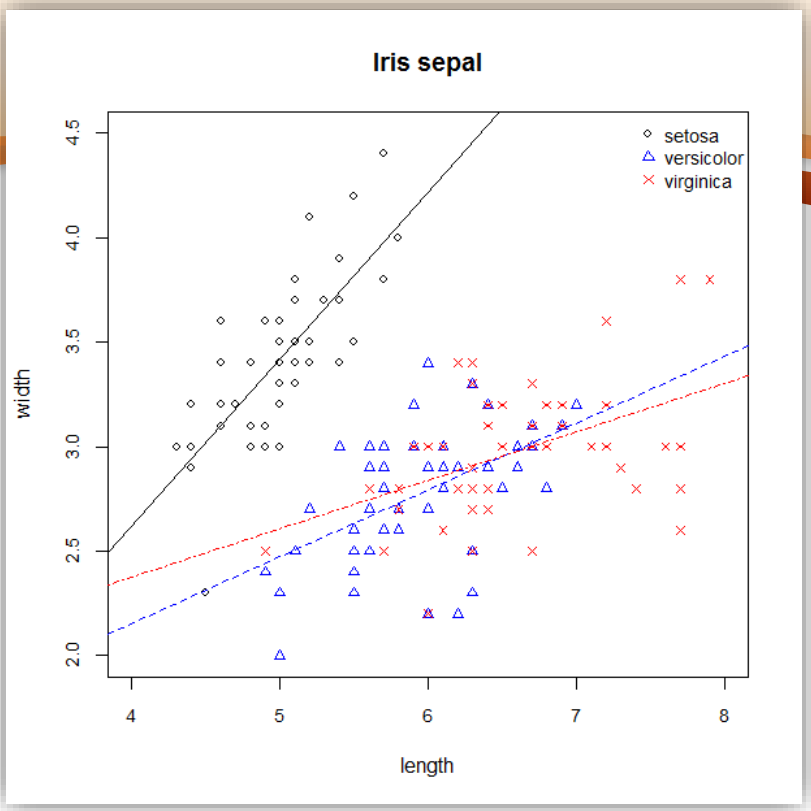
→ more obvious that the body of the function is three lines long

Graphics within R

R graphic functions

- 3 types of R graphic functions:
  - High level: create more or less complete graphs
    Examples: *plot, hist, boxplot, …*
  - Low level: allow additional information to be added to a graph
    Examples: *points, lines, text, axis, …*
  - Interactive graphic functions:
    allow to extract information from an existing graph

Produce Graphic: Steps



1. Produce main graphic
   → high-level plotting functions
   1. R Reference Card
   2. "R Graph Gallery" (website)
      → download code
      → modify and apply it
   3. Google

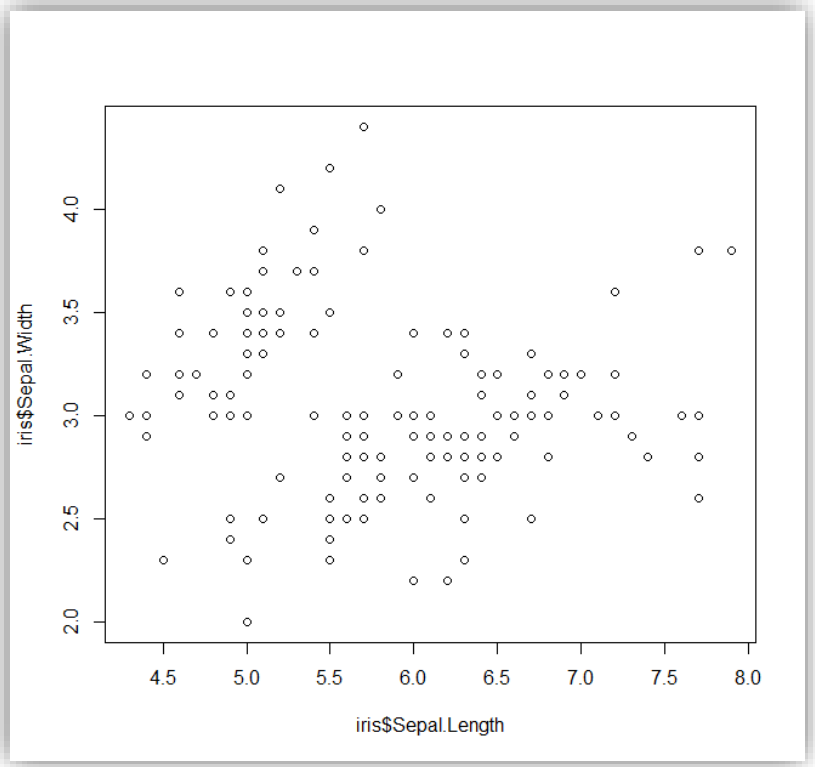
2. Modify graphic parameters (line style, colour, font size, ...)
   1. Use help!

3. Add additional lines, points, title, legends, text,...
   1. R Reference Card
   2. Google

plot()

- Basic high level plotting function: `plot()`
- Plot x-coordinates against y-coordinates: `plot(x,y)`

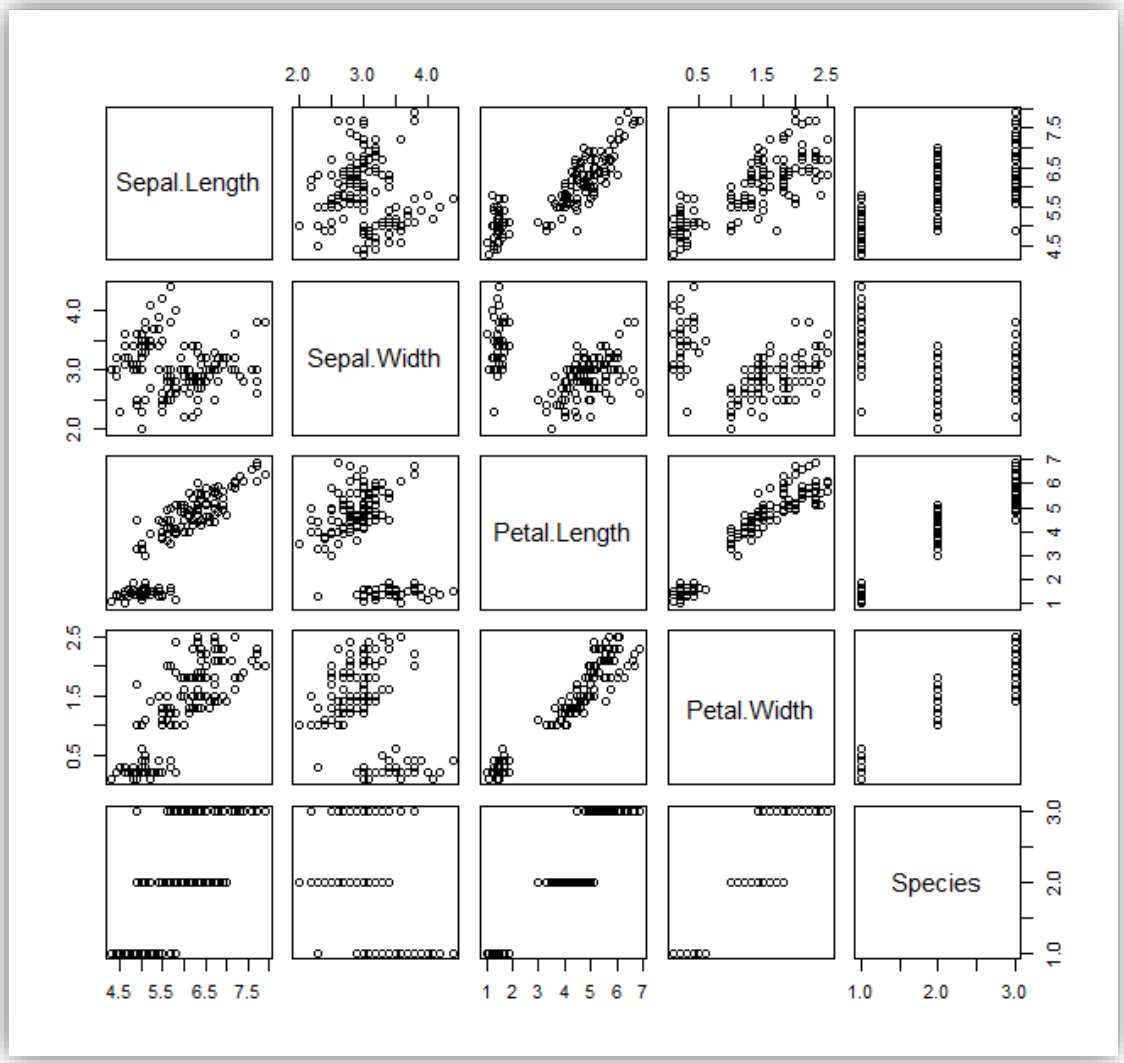
```
plot(iris$Sepal.Length,
     iris$Sepal.Width)
```



- But, plot() is more powerful:
  - What happens if you feed plot a whole dataframe?

```
plot(iris)
```

plot(iris)



plot() – add low level plotting functions

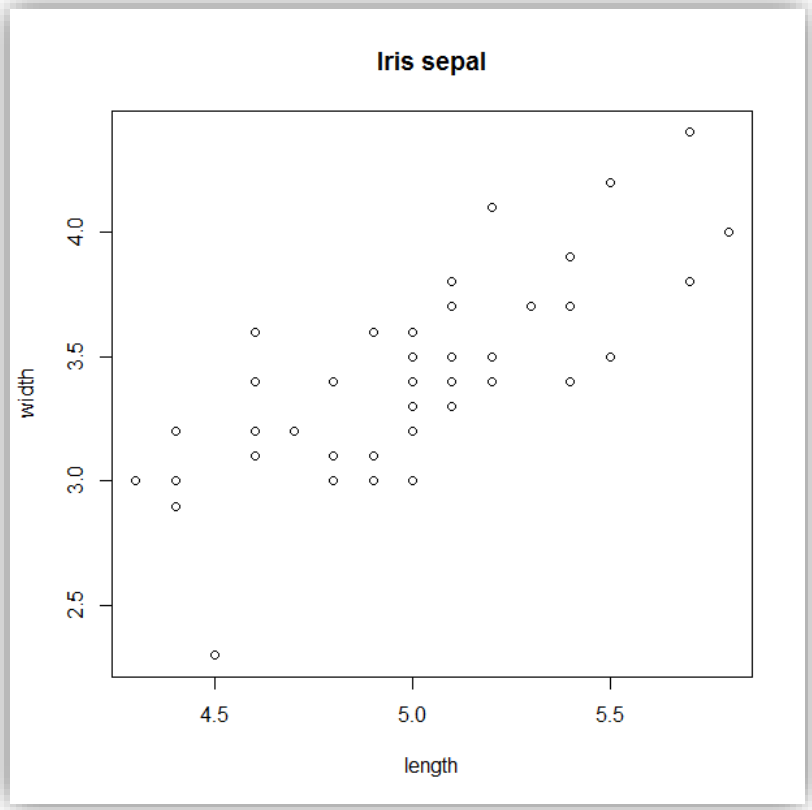
1. Plot sepal length against width of only one species (setosa):

```
irisSe <- iris[iris$Species == "setosa",]
plot(irisSe$Sepal.Length,irisSe$Sepal.Width)
```

Make graphic nicer with low level plotting functions:

2. Add nicer xy-labels (`xlab/ylab`) and title (`main`):

```
plot(irisSe$Sepal.Length,
 irisSe$Sepal.Width,
 xlab="length",ylab="width",
 main="Iris sepal")
```



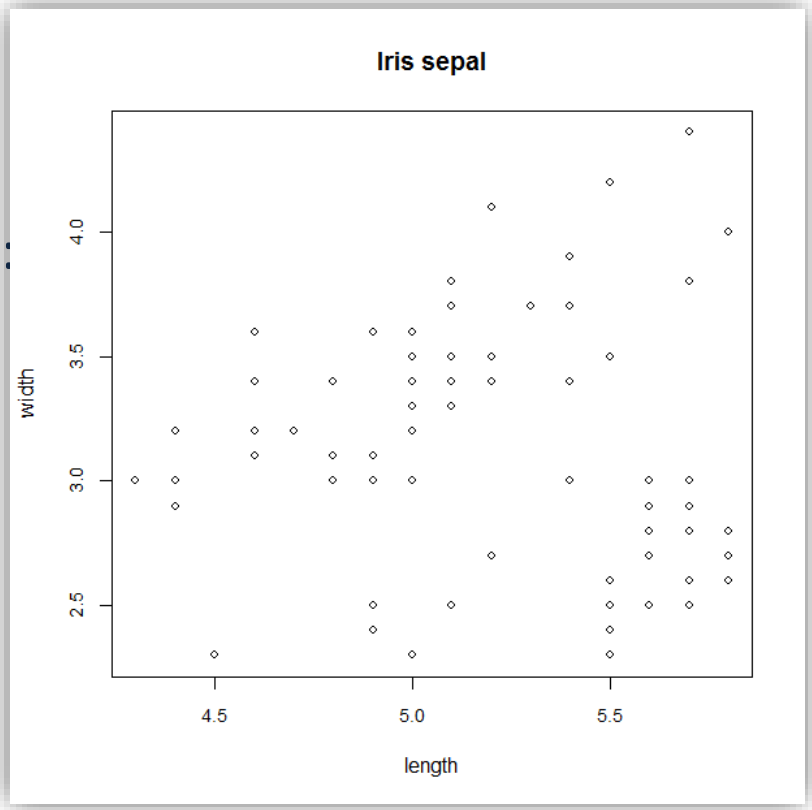
plot() – add low level plotting functions

3. Reduce size of points (`cex`) and axis (`cex.axis`):

```
plot(irisSe$Sepal.Length,
 irisSe$Sepal.Width,
 xlab="length",ylab="width",
 main="Iris sepal",
 cex=0.8, cex.axis=0.9)
```

4. Add data of other species (`points()`):

```
irisVe <- iris[iris$Species
 == "versicolor",]
points(irisVe$Sepal.Length,
 irisVe$Sepal.Width,cex=0.8)

irisVi <- iris[iris$Species
 == "virginica",]
points(irisVi$Sepal.Length,
 irisVi$Sepal.Width,cex=0.8)
```



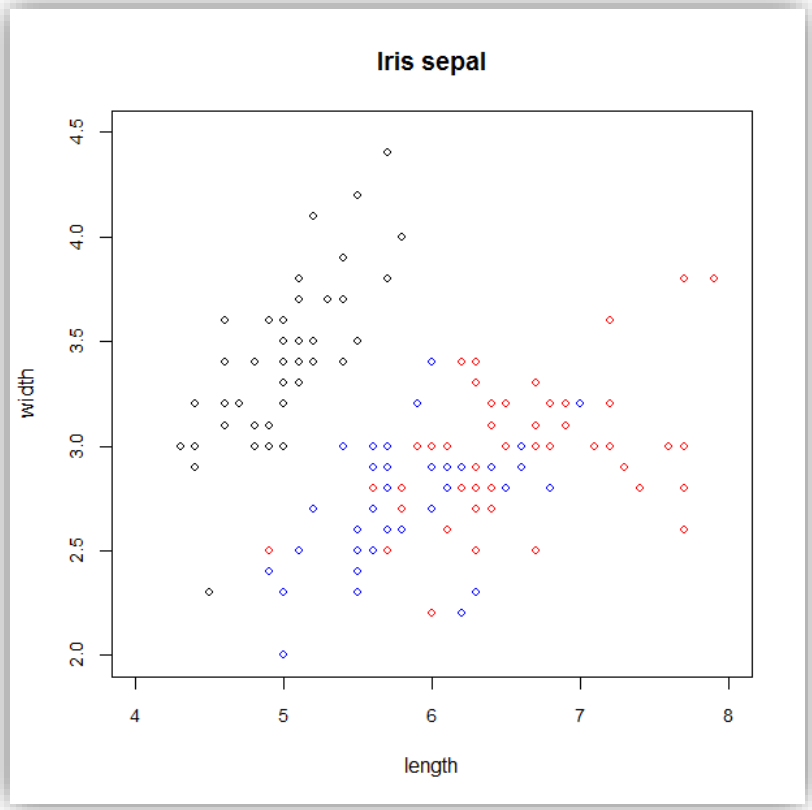
plot() – add low level plotting functions

5. Adjust axis range (`ylim/xlim`):

```
plot(irisSe$Sepal.Length,
 irisSe$Sepal.Width,
 xlab="length",ylab="width",
 main="Iris sepal",
 cex=0.8,cex.axis=0.9,
 xlim=c(4,8),ylim=c(2,4.5))
```

6. Color points according species (`col`):

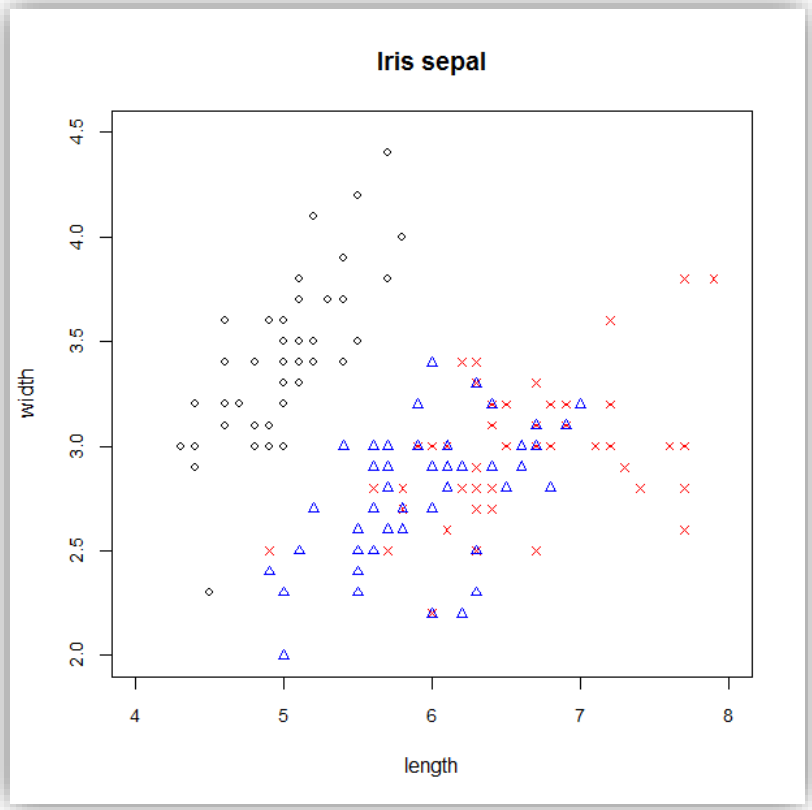
```
points(irisVe$Sepal.Length,
 irisVe$Sepal.Width,cex=0.8,
 col="blue")

points(irisVi$Sepal.Length,
 irisVi$Sepal.Width,cex=0.8,
 col="red")
```



plot() – add low level plotting functions

7. Adapt symbol of the points according species (`pch`):

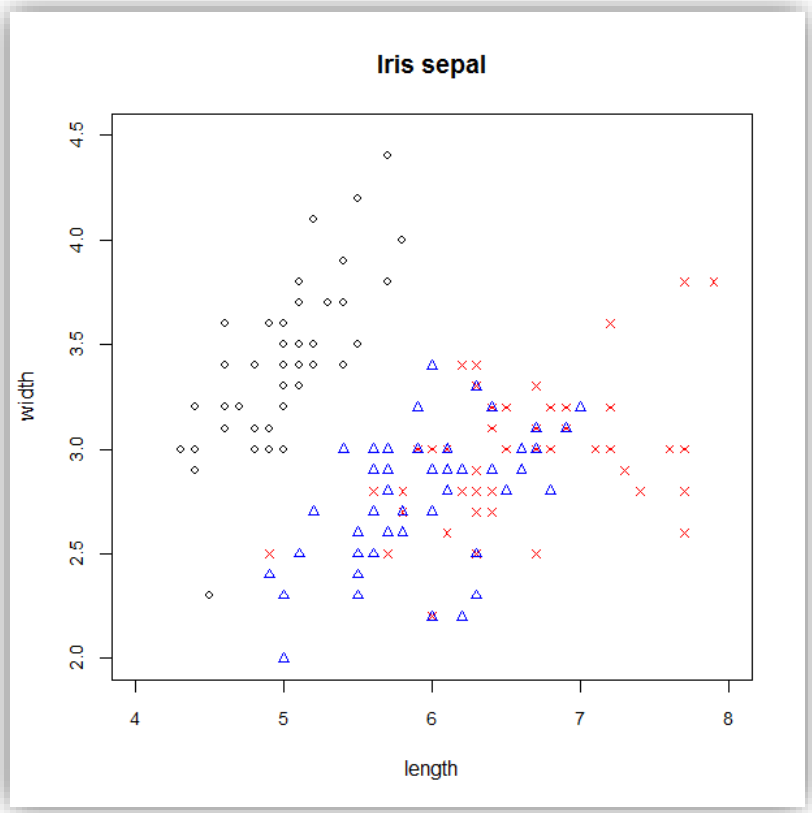

```
points(irisVe$Sepal.Length,
  irisVe$Sepal.Width,cex=0.8,
  col="blue",pch=2)

points(irisVi$Sepal.Length,
  irisVi$Sepal.Width,cex=0.8,
  col="red",pch=4)
```



plot() – add low level plotting functions

8. Add legend (`legend()`):

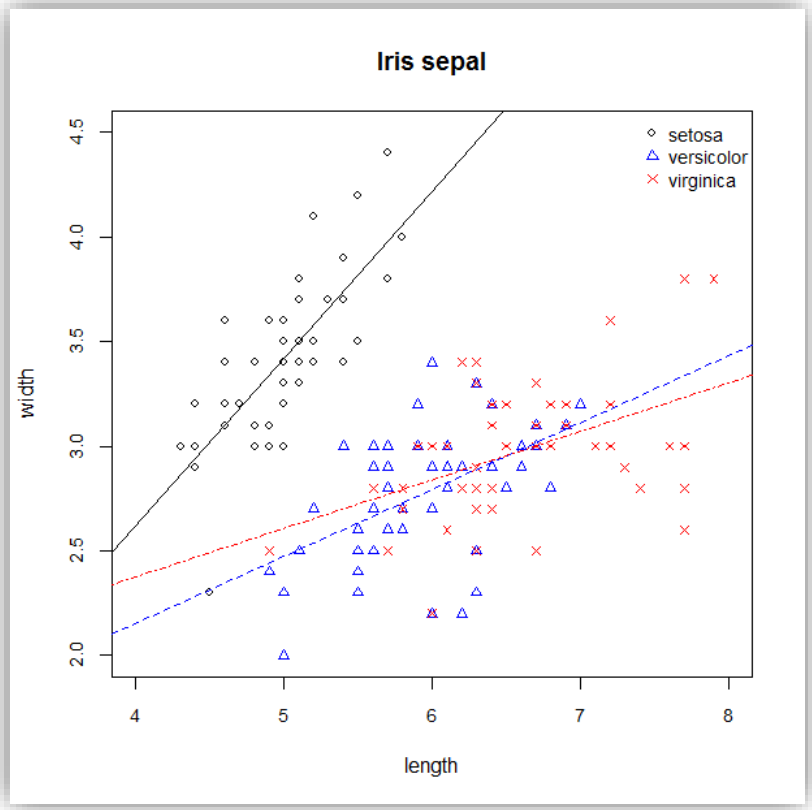
   Try to add a legend to the plot, use `?legend` as help



plot() – add low level plotting functions

9. Add regression lines:
   - lm(): fit a linear model
   - abline(): adds a straight line to the plot
   - Adjust line type (`lty`):
     1:solid, 2:dashed, 3:dotted,
     4:dotdash, 5:longdas, 6:twodash

```
abline(lm(irisSe$Sepal.Width
 ~irisSe$Sepal.Length))

abline(lm(irisVe$Sepal.Width
 ~irisVe$Sepal.Length),
 col="blue",lty=2)

abline(lm(irisVi$Sepal.Width
 ~irisVi$Sepal.Length),
 col="red",lty=4)
```



Structure of R objects

Side note: Display the structure of an R object `str(object)`

```
regLine <- lm(irisSe$Sepal.Width~irisSe$Sepal.Length)
regLine
Call:
lm(formula = irisSe$Sepal.Width ~ irisSe$Sepal.Length)

Coefficients:
        (Intercept)  irisSe$Sepal.Length
            -0.5694                0.7985


str(regLine)
List of 12
 $ coefficients : Named num [1:2] -0.569 0.799
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "irisSe$Sepal.Length"
 $ residuals    : Named num [1:50] -0.00306 -0.34336 0.01635 -0.0038 ...
 ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
...
```

Access intercept: `regLine$coefficients[1]`

Save plot

- Saves a plot in png format (see `?png` for additional options):

```
png("irisPlot.png")
 plot(...)
 points(...)
 ...
dev.off()
```

opens graphic device

closes graphic device

- Saves a plot as pdf (see `?pdf` for additional options):

```
pdf("irisPlot.pdf")
 plot(...)
 points(...)
 ...
dev.off()
```

Try to save your plot

Histogram

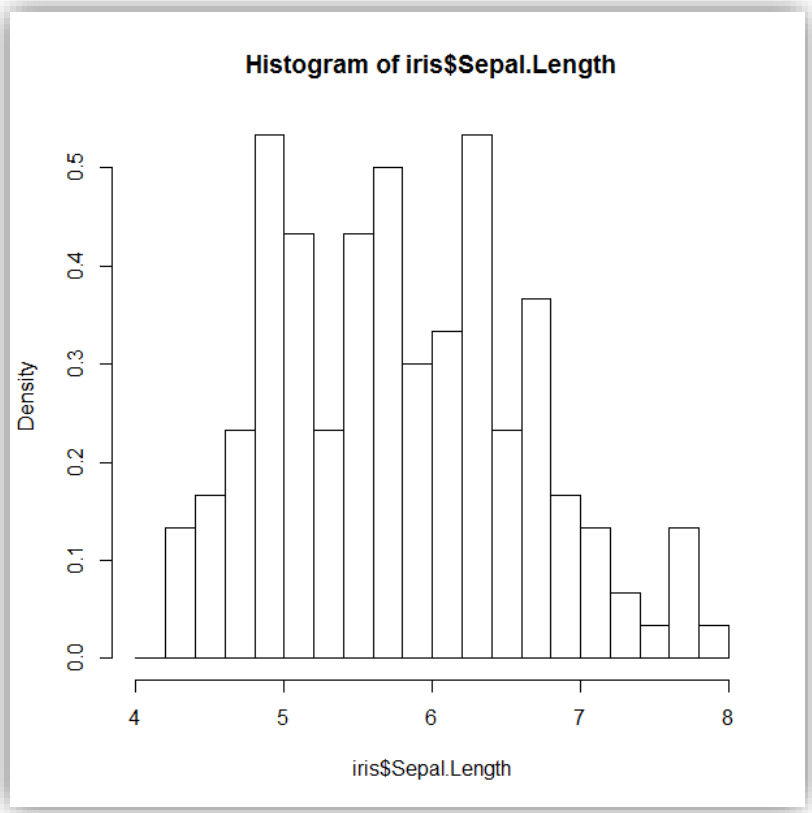
- Histogram: `hist()`

```
hist(iris$Sepal.Length)
```

- Divide data into more bins by 'breaks':
  e.g.: `breaks=20` or
  `breaks=c(4,4.5,5,5.5,6)`

- Plotting density instead of frequency:
  frequency divided by the width of the interval (plot area sums to 1)

```
hist(iris$Sepal.Length,
 breaks=seq(4,8,by=0.2),
 freq=FALSE)
```



Histogram

- Plot only data of setosa and change axis labels and title

Histogram

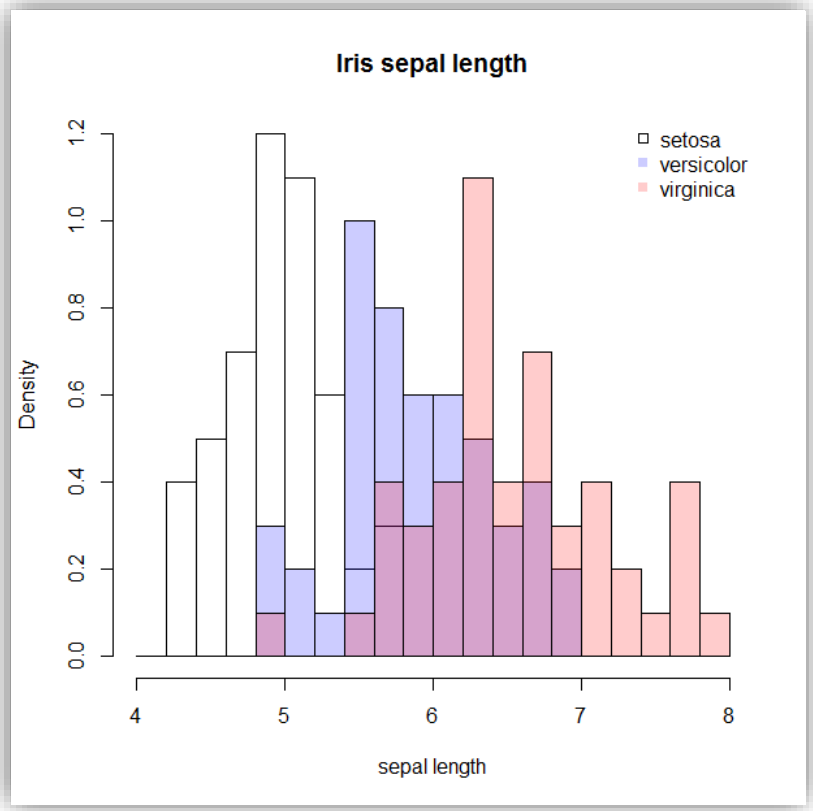
- Overlay data of other species:
  - use argument: `add=TRUE`
  - Don’t forget to color it and add a legend

Histogram

3. Make colors transparent:
   - Use rgb function: `rgb(red, green, blue, alpha)`

intensity — transparency

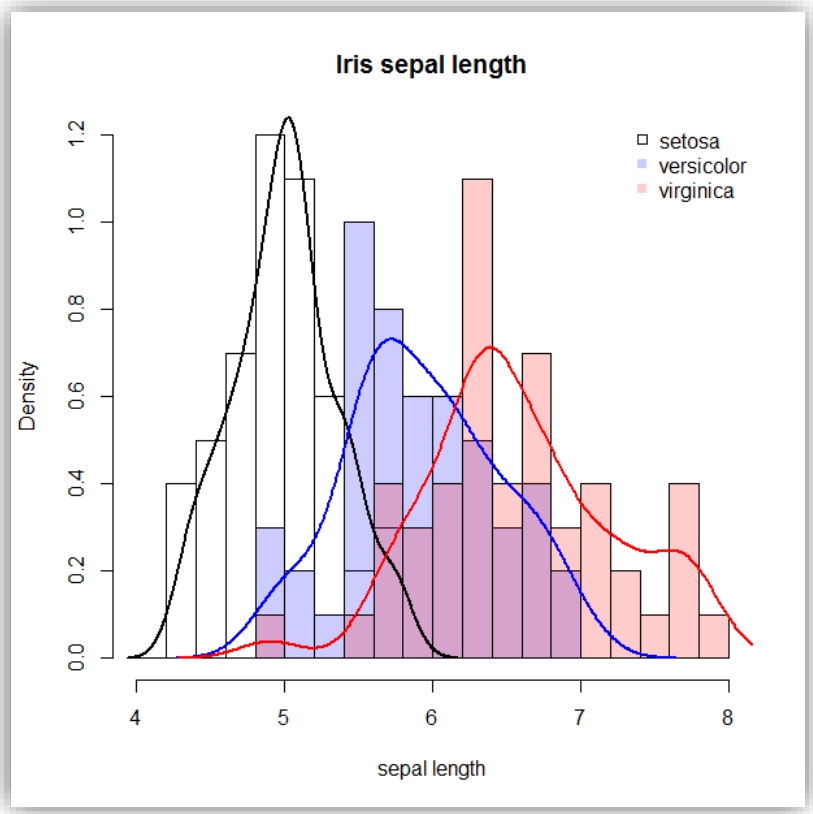
```
blueT <- rgb(0,0,1,0.4)
redT <- rgb(1,0,0,0.4)

hist(irisVi$Sepal.Length,
 breaks=seq(4,8,by=0.2),
 freq=FALSE,add=TRUE,col=blueT)

hist(irisVe$Sepal.Length,
 breaks=seq(4,8,by=0.2),
 freq=FALSE,add=TRUE,col=redT)

legend("topright",c("setosa",
 "versicolor","virginica"),
 col=c("black",blueT,redT),
 pch=c(0,15,15),bty="n")
```



Histogram

4. Add a fitted density curve
   - `density()`: computes kernel density estimates
   - `lines()`: adds a connected line

```
lines(density(irisSe$Sepal.Le
 ngth),lwd=2)

lines(density(irisVe$Sepal.Le
 ngth),lwd=2,col="blue")

lines(density(irisVi$Sepal.Le
 ngth),lwd=2,col="red")
```



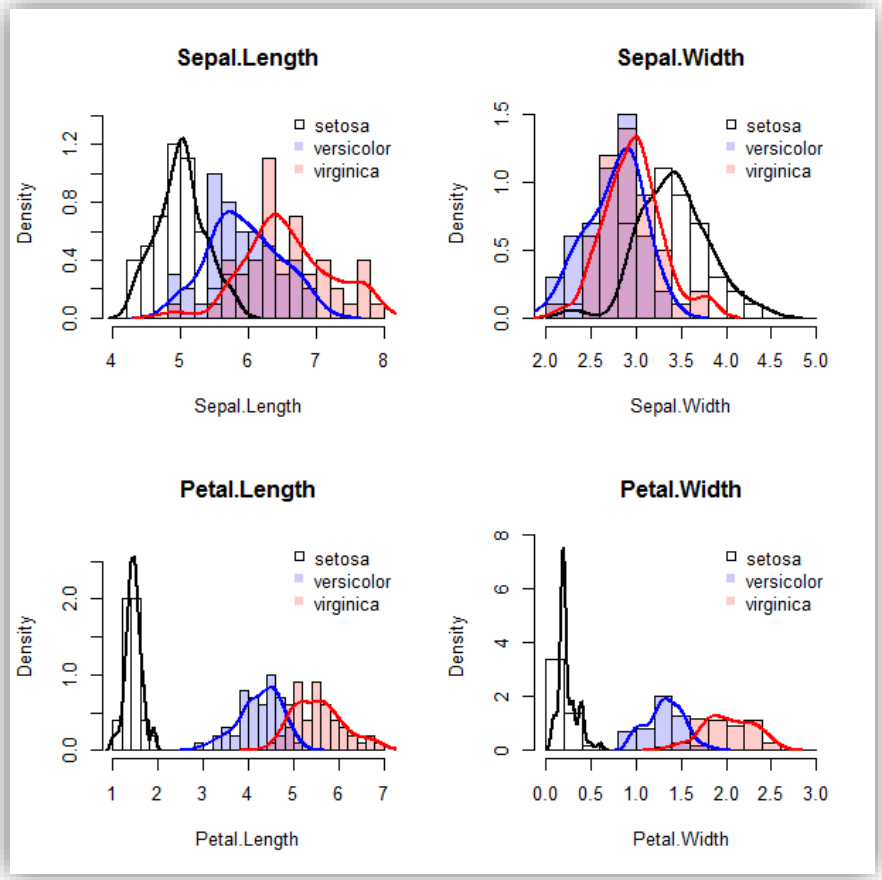
Multiple plots

5. Put all 4 measurements in one graphic → 4 plots in one graphic
   - Change `mfrow` within graphic parameters (par)

store old settings

```
old.par <- par(mfrow=c(2, 2))
 hist(irisSe$Sepal.Width,...
 ...
 hist(irisSe$Sepal.Length,...
 ...
 hist(irisSe$Petal.Length, ...
 ...
 hist(irisSe$Petal.Width, ...
...
par(old.par)
```

reset old settings



Multiple plots

5. Put all 4 measurements in one plot → by using for-loop

```
old.par <- par(mfrow=c(2, 2))
for(i in 1:(ncol(irisSe)-1)){
 colName <- colnames(iris)[i]
 xmin <- trunc(min(iris[,i]))
 xmax <- trunc(max(iris[,i]))+1
 ymax <- max(density(irisSe[,i])$y, density(irisVe[,i])$y,
      density(irisVi[,i])$y)+0.2
 hist(irisSe[,i],breaks=seq(xmin,xmax,by=0.2),ylim=c(0,ymax),
      freq=FALSE,main=colName,xlab=colName)
 hist(irisVe[,i],breaks=seq(xmin,xmax,by=0.2),freq=FALSE,add=TRUE,
      col=blueT)
 hist(irisVi[,i],breaks=seq(xmin,xmax,by=0.2),freq=FALSE,add=TRUE,
      col=redT)
 legend("topright",c("setosa","versicolor","virginica"),
      col=c("black",blueT,redT),pch=c(0,15,15),bty="n")
 lines(density(irisSe[,i]),lwd=2)
 lines(density(irisVe[,i]),lwd=2,col="blue")
 lines(density(irisVi[,i]),lwd=2,col="red")
}
par(old.par)
```

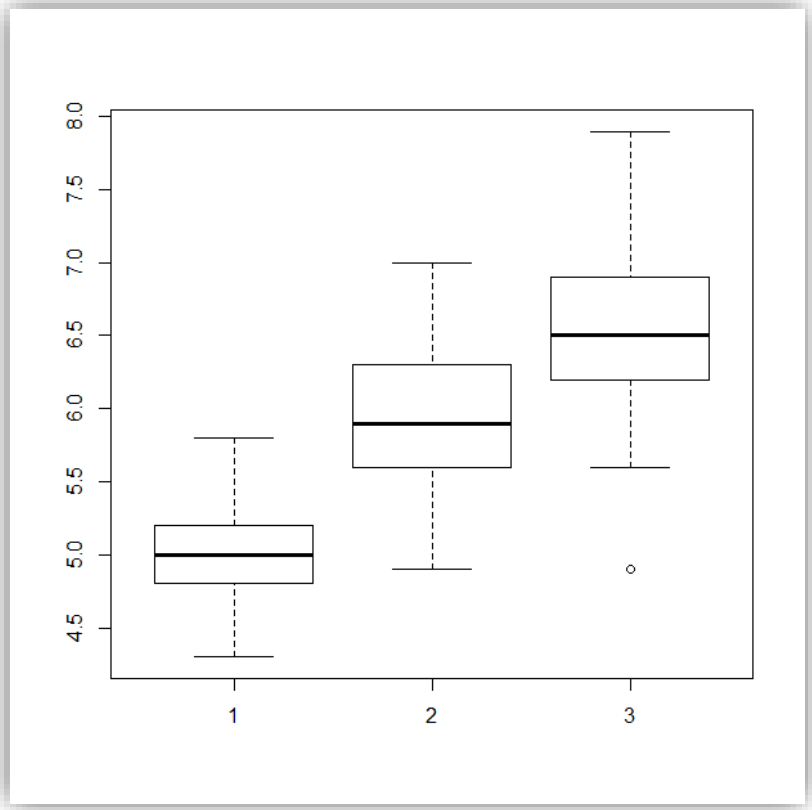
boxplot

- Boxplots: `boxplot()`

```
boxplot(iris$Sepal.Length)
```

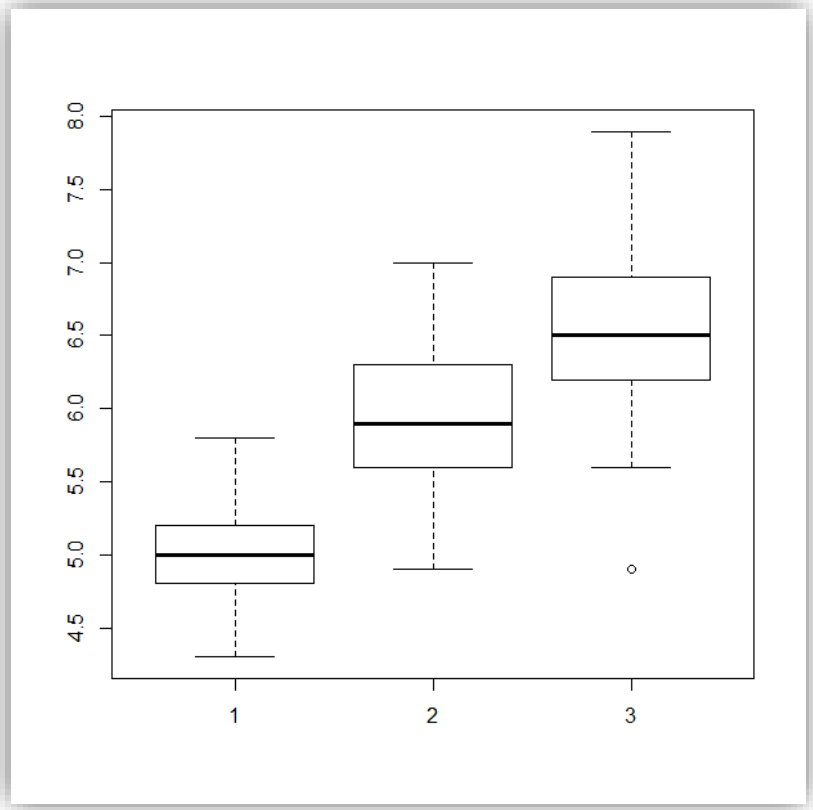
- More than one boxplot in the graphic:

```
boxplot(irisSe$Sepal.Length,
  irisVe$Sepal.Length,
  irisVi$Sepal.Length)
```



boxplot

- Change axis labels, title and color boxplots
  (use `?boxplot` if necessary)



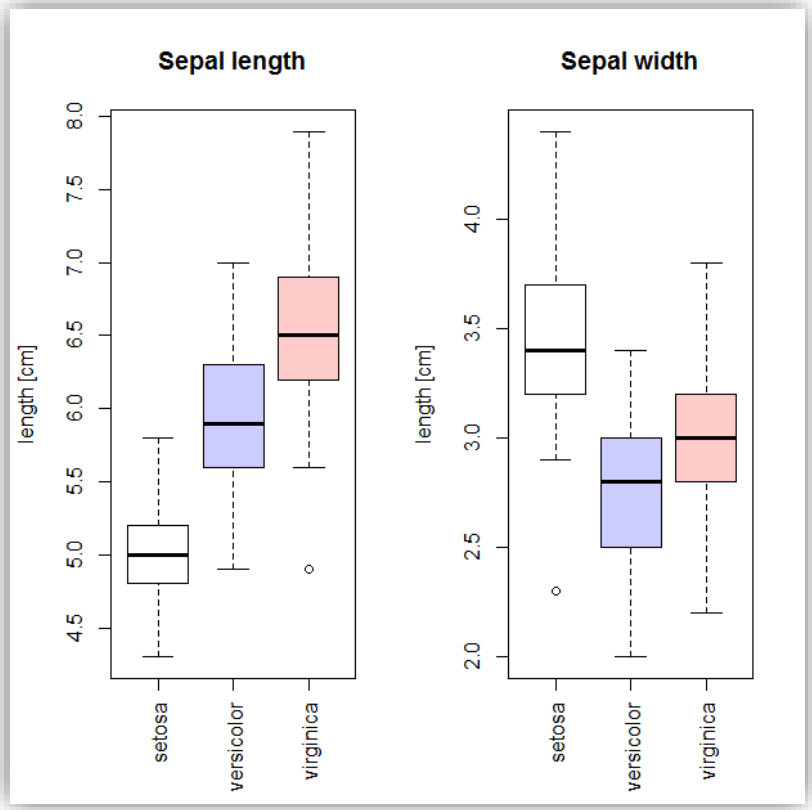
boxplot

- Try to put two boxplot graphics (sepal length/width) into one graphic

boxplot

- Write axis labels always vertical: `las=3` (see `?par` for details)

```
old.par <- par(mfrow=c(1,2))
 boxplot(irisSe$Sepal.Length,
  irisVe$Sepal.Length,
  irisVi$Sepal.Length,
  ylab="length [cm]",
  names=c("setosa","versicolor",
  "virginica"),main="Sepal length",
  col=c("white",blueT,redT),las=3)
 boxplot(irisSe$Sepal.Width,
  irisVe$Sepal.Width,
  irisVi$Sepal.Width,
  ylab="length [cm]",
  names=c("setosa","versicolor",
  "virginica"),main="Sepal width",
  col=c("white",blueT,redT),las=3)
par(old.par)
```



barplot

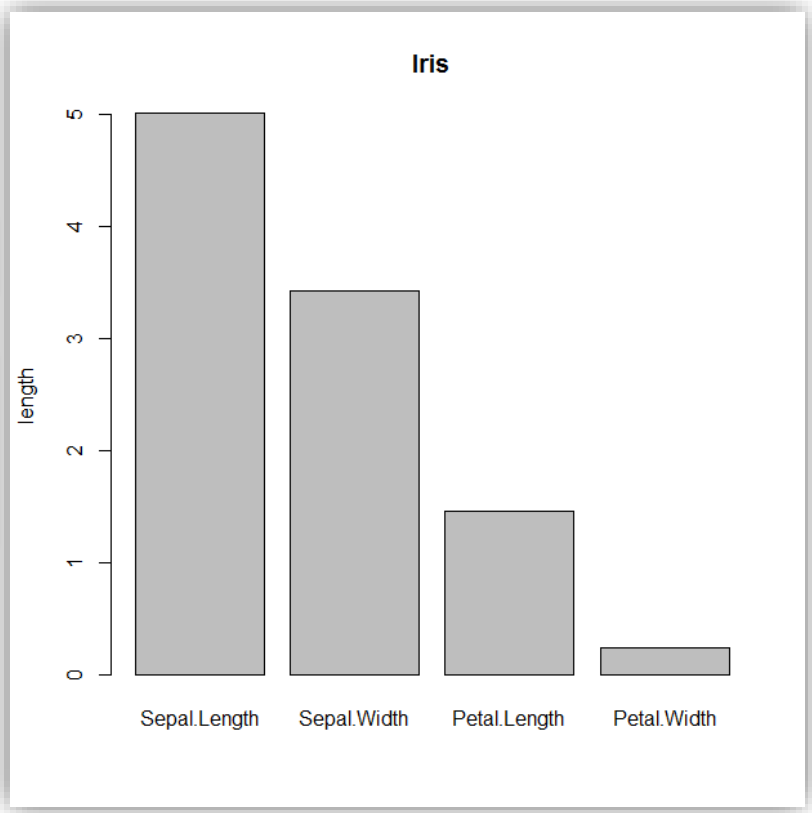
- Barplots: `barplot()`

```
barplot(c(mean(irisSe$Sepal.Length),
          mean(irisSe$Sepal.Width)))
```

- Use `sapply()` to estimate mean for each columns:

```
a <- sapply(irisSe[,1:4],mean)
```

```
barplot(a,ylab="length",
 names=colnames(irisSe[,1:4]),
 main="Iris setosa")
```



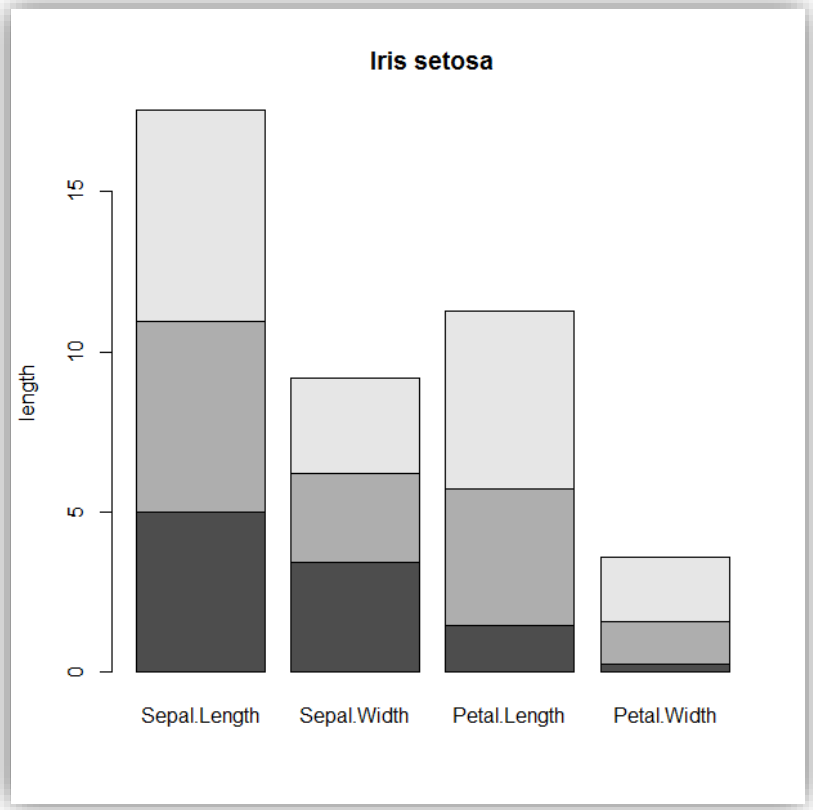
barplot

- Add data of other species:
  - Feed matrix instead vector: columns: mean measurements
    rows: species

```
means <- matrix(
 c(sapply(irisSe[,1:4],mean),
 sapply(irisVe[,1:4],mean),
 sapply(irisVi[,1:4],mean)),
 ncol=4, byrow=TRUE)
```

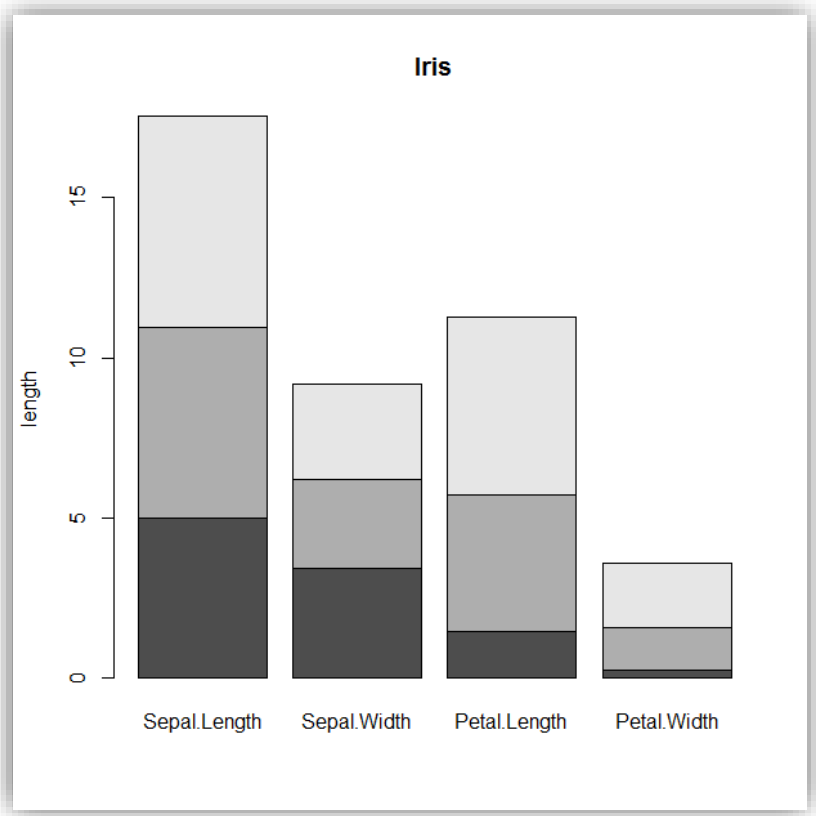
```
     [,1]  [,2]  [,3]  [,4]
[1,] 5.006 3.428 1.462 0.246
[2,] 5.936 2.770 4.260 1.326
[3,] 6.588 2.974 5.552 2.026
```

```
barplot(means,ylab="length",
 names=colnames(irisSe[,1:4]),
 main="Iris setosa")
```



barplot

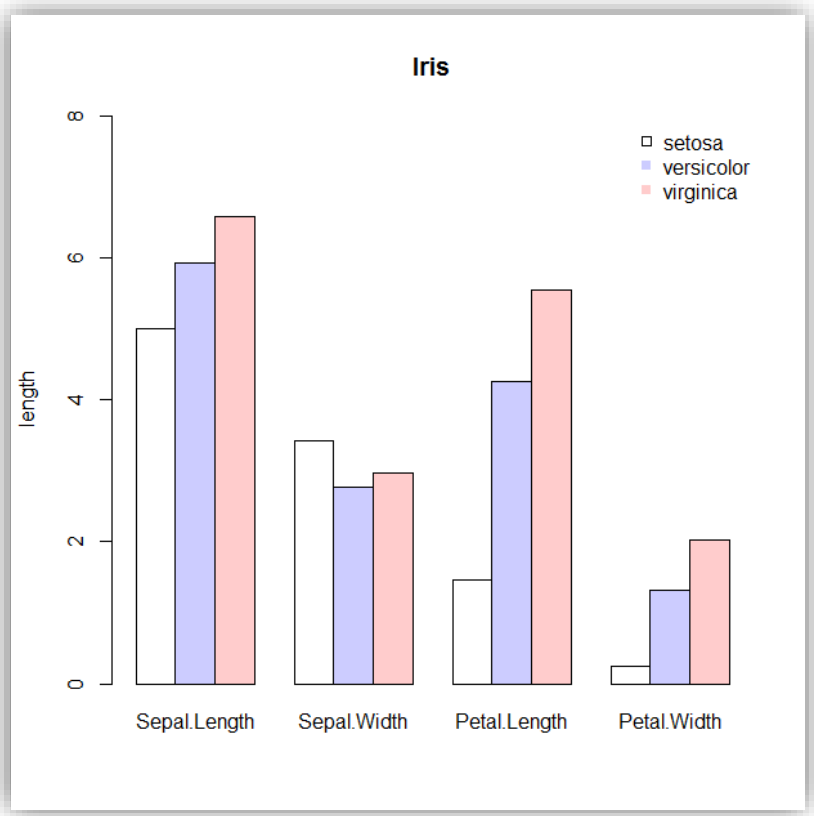
- Use `beside=TRUE` to plot bars beside each other, color bars according the color scheme and add legend



barplot

- Add standard deviations to the bars:
  1. Create matrix with sd:

```
stDev <- matrix(
 c(sapply(irisSe[,1:4],sd),
 sapply(irisVe[,1:4],sd),
 sapply(irisVi[,1:4],sd)),
 ncol=4, byrow=TRUE)
```



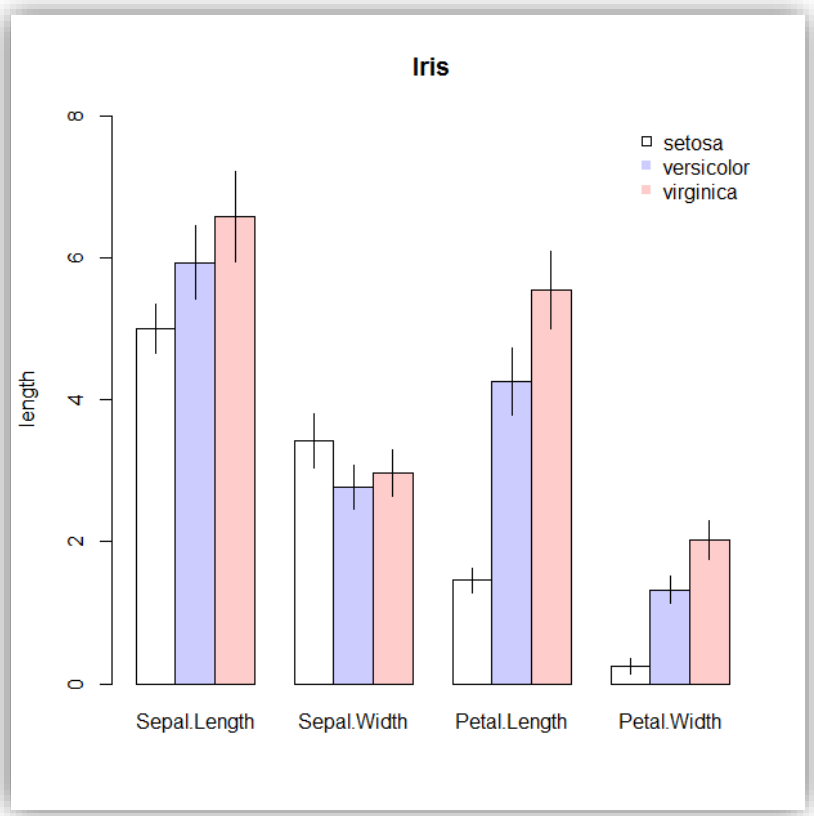
barplot

2. Save “bar positions” in a variable

```
bplot <- barplot(means,
 ylab="length",names=colnames(
 irisSe[,1:4]),
 main="Iris",beside=TRUE,
 col=c("white",blueT,redT),
 ylim=c(0,8))
```

3. Add the sd bars: `segments()`

```
segments(bplot,means-stDevs,
         bplot,means+stDevs)
```



heatmap

- heatmaps:
  - `image()`
  - `levelplot()` → creates automatically a legend

- We need a matrix data set: relative sepal length across each sample:

```
matrixSe <- matrix(NA,nrow(irisSe),nrow(irisSe))
for(i in 1:nrow(matrixSe)){
  matrixSe[i,] <- irisSe$Sepal.Length[i]/irisSe$Sepal.Length
}
```

heatmap

Try to create an `image()` and a `levelplot()` (use help if necessary)

→ don't forget to add a nice title and axis labels

→ use `colRmp <- colorRampPalette(c("color1","color2",…))(nbBreaks)` to create your own color ramp

Acknowledgment

- Tutorial with solutions:
  https://www.dropbox.com/s/kath5t9q1p8cyj1/URPP_Tutorial_RBasics_1_v2_solutions.pdf

- Sources:
  - http://nicercode.github.io/intro/
  - http://www.cyclismo.org/tutorial/R/
  - http://manuals.bioinformatics.ucr.edu/home/programming-in-r#TOC-Control-Structures
  - http://streaming.stat.iastate.edu/workshops/r-intro/lectures/5-Rprogramming.pdf

- R reference cards:
  - http://cran.r-project.org/doc/contrib/Short-refcard.pdf
  - http://cran.r-project.org/doc/contrib/Baggott-refcard-v2.pdf

- Graphic tutorials:
  - http://ww2.coastal.edu/kingw/statistics/R-tutorials/graphically.html
  - http://www.statmethods.net/graphs/creating.html
  - http://bio.fsu.edu/miller/docs/Tutorials/Tutorial3_Graphics.pdf
  - http://www.harding.edu/fmccown/r/
  - http://manuals.bioinformatics.ucr.edu/home/R_BioCondManual#TOC-Graphical-Procedures

- R graph gallery:
  - http://rgraphgallery.blogspot.ch/
  - http://www.sr.bham.ac.uk/~ajrs/R/r-gallery.html