R Notebook

R data-types and structures

Variable

A named container or storage that is used to reference a value in the code.

- This value can be of many types:
  - numeric: `1, 2, 3.4`
  - character: `"a", "book", "c"`
  - logical: `TRUE, FALSE` or simply `T, F`
  - Integers, complex, and factors

Variable

- Constraints on names:
  - must starts with a letter or a dot not followed by a number.
  - can only contain letters, numbers, underscore and the dot character "."

  `var1, .var1, var.name, v1`

  `1v, .1v, var-1, var#1`

Variable

`<-` and `=` are used for assignment of values in R

```
x <- 2
y <- 3
x+y
```

```
## [1] 5
```

- Note that R variables are dynamically typed

```
x <- 2
x <- "hello"
```

Vector

A `vector` is the simplest data structure in R. it represents a container for a set of elements of the same data type. - Examples:

```
c(1,3,5)
```

```
## [1] 1 3 5
```

```
length(c("test"))
```

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

```
class(c(TRUE,TRUE,FALSE))
```

```
## [1] "logical"
```

Vector

`c` function is used to concatenate vectors.

```
x <- c(1,3,5)
y <- c(2,4,6)
z <- c(x,y)
z
```

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

Remember: vectors have a single data type

```
x <- c(1,3,5)
y <- c("a","b")
c(x,y)
```

```
## [1] "1" "3" "5" "a" "b"
```

Vector

Arithmetics are performed on vectors member-by-member

```
x <- c(1:3) # R way to create a sequence of numbers
x
```

```
## [1] 1 2 3
```

```
x + 1
```

```
## [1] 2 3 4
```

Vector

and this extends to operations between vectors

```
x <- c(1,3,5)
y <- c(2,4,6)
x+y
```

```
## [1]  3  7 11
```

Vector

What will happen if the two vectors are of different lengths?

```
x <- c(1,3,5)
y <- c(2,4,6,8,10)
z <- x+y
```

Vector

What will happen if the two vectors are of different lengths?

```
x <- c(1,3,5)
y <- c(2,4,6,8,10)
z <- x+y
z
```

```
## [1]  3  7 11  9 13
```

Vector

Exercise: Create a vector of Logical values, assign it to a variable. Create a vector of Integers and assign it to a variable.

Try subtracting the two vectors. What happened?

Vector

Exercise: Create a vector of Logical values, assign it to a variable. Create a vector of Integers and assign it to a variable.

Try subtracting the two vectors. What happened?

```
x <- c(T,F,T)
y <- c(3,4,5)
x-y
```

```
## [1] -2 -4 -4
```

logical values in R can be coerced to integers, i.e. `1` for `TRUE` and `0` for `FALSE`

Vector

Retrieving elements from a vector is done by `indexing`

- There are many ways to use indexing and the simplest is `numeric indexing`, i.e. a number that represents the position of an element in a vector.
- `[]` is used for indexing. In R, indexing starts from `1`

```
  x <- c(1,3,5,7,9)
  x[2]
```

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

```
  x[c(3,5)]
```

```
## [1] 5 9
```

Vector - Indexing

`indexing` can be used also for changing values in a vector

```
  x <- c(1,3,5,7,9)
  x[2] <- 4
  x
```

```
## [1] 1 4 5 7 9
```

```
  x[c(3,5)] <- c(6,10)
  x
```

```
## [1]  1  4  6  7 10
```

```
  x[-1] # negative indexing is used to remove elements
```

```
## [1]  4  6  7 10
```

Vector - Indexing

Logical values can be used for `indexing` too.

```
  x <- c(1,3,5,7,9)
  x[c(T,F,T,F,T)]
```

```
## [1] 1 5 9
```

```
  x[c(F,F,T,F,T)] <- c(6,10)
  x
```

```
## [1]  1  3  6  7 10
```

```
  y <- x[c(T,F)] # what will happen??
```

Vector - Indexing

Logical values can be used for `indexing` too.

```
  x <- c(1,3,5,7,9)
  y <- x[c(T,F)] # what will happen??
  y
```

```
## [1] 1 5 9
```

Named Vectors

R allows us to give names to each element in a vector, and use them to index:

```
x <- c(2,4,6,8,10)
names(x) <- c("A", "B", "C", "D", "E")
x
```

```
##  A  B  C  D  E
##  2  4  6  8 10
```

```
x[c("A", "D")]
```

```
## A D
## 2 8
```

Matrix

A `matrix` is a collection of data elements arranged in a two-dimensional rectangular layout

```
x = matrix(
   c(2, 4, 3, 1, 5, 7), # the data elements
   nrow=2,              # number of rows
   ncol=3)             # number of columns
x
```

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

```
dim(x)  # return the dimensions of a matrix; #rows and #columns
```

```
## [1] 2 3
```

```r
x[2,3] # note the ',' to indiacte row and column indexing
```

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

Matrix - Names

Matrix can have row and/or column names:

```r
x <- matrix(data=1:10, nrow=2, ncol=5)
rownames(x) <- c("A", "B")
x
```

```
##   [,1] [,2] [,3] [,4] [,5]
## A    1    3    5    7    9
## B    2    4    6    8   10
```

```r
x["B", c(2,5)]
```

```
## [1]  4 10
```

```r
x["A",]
```

```
## [1] 1 3 5 7 9
```

Dataframe

A `dataframe` is used for storing data tables. It represents a list of vectors of equal length and can store vectors of different types.

```r
patientsID = c("p1", "p2", "p3")
age = c(25, 35, 55)
alive = c(TRUE, FALSE, NA)
df = data.frame(patientsID, age, alive)       # df is a data frame
df
```

```
##   patientsID age alive
## 1         p1  25  TRUE
## 2         p2  35 FALSE
## 3         p3  55    NA
```

Dataframe

$ is another way to index columns in a named data.frame

```r
df[,"alive"]
```

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

```r
df$alive
```

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

Dataframe

```r
patientsID = c("p1", "p2", "p3")
age = c(25, 35, 55)
alive = c(TRUE, FALSE, NA)
df = data.frame(patientsID, age, alive)       # df is a data frame
df$patientsID  # factor type, we will talk about it later
```

```
## [1] "p1" "p2" "p3"
```

```
df = data.frame(patientsID, age, alive, stringsAsFactors = F)
df$patientsID
```

```
## [1] "p1" "p2" "p3"
```

List

A `list` is a generic vector that can contain multiple data types. Think of it as a vector of boxes, you can put anything in the box.

```
x <- list(1, "a", c(TRUE, FALSE))
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1]  TRUE FALSE
```

List - indexing

One square bracket `[]` will retrieve the box, while double square brackets would unpack the box

```
patientsID = c("p1", "p2", "p3", "p4", "p5")
age = c(25, 35, 55)
alive = c(TRUE, FALSE, TRUE, NA, FALSE)
x <- list(patientsID,age,alive)
x[2]
```

```
## [[1]]
## [1] 25 35 55
```

```
x[[2]]
```

```
## [1] 25 35 55
```

List - indexing

```
patientsID = c("p1", "p2", "p3", "p4", "p5")
age = c(25, 35, 55)
x <- list(patientsID,age)
x[[2]][3]
```

```
## [1] 55
```

List - indexing

Named lists provide another way to index them using the operator `$`

```
patientsID = c("p1", "p2", "p3", "p4", "p5")
age = c(25, 35, 55)
x <- list("sampleID"=patientsID,"age"=age)
x$age
```

```
## [1] 25 35 55
```

```r
x[["sampleID"]]
```

```
## [1] "p1" "p2" "p3" "p4" "p5"
```

R data-types and structures

These are the main data types and structures in R.

We still need to talk about factors!

R operations

R Arithmetic operators

```r
x <- 5
y <- 3
+ x        # 5
- x        # -5
x + y      # 8
x - y      # 2
x * y      # 15
x / y      # 1.666667
x ^ y      # 125 , x**y is the same
x %% y     # 2
x %/% y    # 1
```

R Relational operators

```r
x <- 5
y <- 3
x < y     # FALSE
x > y     # TRUE
x <= y    # FALSE
x >= y    # TRUE
x == y    # FALSE
x != y    # TRUE
```

R Logical operators

```r
x <- TRUE
y <- FALSE
!x        # FALSE  -- NOT
x && y    # FALSE  -- Logical AND
x || y    # TRUE   -- Logical OR
```

```r
x <- c(TRUE,TRUE,FALSE)
y <- c(TRUE,FALSE,FALSE)
x & y   # element-wise AND
```

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

```
x | y   # element-wise OR
```

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

R control flow structures

if-else

This is a conditional flow structure by which the following steps of the program are dependent on the outcome of the condition.

```
if(3>2){
  print("Yes")
}
```

```
## [1] "Yes"
```

```
if(3<2){
  print("Yes")
}else{
  print("No")
}
```

```
## [1] "No"
```

if-else

```
if(3<2){
  print("No")
}else if(3>2){
  print("Yes")
}else{
  print("Maybe")
}
```

```
## [1] "Yes"
```

if-else

You can make the condition more complex

```
x <- 5; y <- "food"
if(x<2 && y == "drink"){
  print("Yes")
}else{
  print("No")
}
```

```
## [1] "No"
```

Loops

Loops are used to repeat a block of code until a condition is satisfied

There are two flow structures for loops, i.e., `for` loops and `while` loops

for loops

The most common use is to iterate through the elements of a vector

```
myVec <- c("A", "B", "C")
for(letter in myVec){
  print(letter)
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
```

for loops

One can also iterate through a vector using its indices

```
myVec <- c(2, 5, 21)
for(i in 1:length(myVec)){
  print(myVec[i] + 3)
}
```

```
## [1] 5
## [1] 8
## [1] 24
```

Note: `seq_along(myVec)` is a built-in function that might be a better option than using `1:length(myVec)`. The latter could break when `length(myVec)` is accidentally 0.

while loops

Repeats executing the code enclosed while a condition is satisfied

```
myVec <- c("a","b","c")
i <- 1
while(i <= length(myVec)){
  print(myVec[i])
  i <- i + 1
}
```

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

Exercise

Create a R object from these vectors and return patients who are females and under the age of 30

```
patientsID <- c("p1", "p2", "p3", "p4", "p5")
age <- c(15, 32, 44, 25, 39)
gender <- c("F","M","M","F","F")
```

Exercise

Create a R object from these vectors and return patients who are females and under the age of 30

```
patientsID <- c("p1", "p2", "p3", "p4", "p5")
age <- c(15, 32, 44, 25, 39)
gender <- c("F","M","M","F","F")
```

```
df <- data.frame(patientsID,age,gender,stringsAsFactors = F)
df[df$gender=="F" & df$age < 30,]
```

```
##   patientsID age gender
## 1         p1  15      F
## 4         p4  25      F
```

Exercise

Create a R object from these vectors and change vector `insure` to `T` for each patient who is a female and above the age of 20

```
patientsID <- c("p1", "p2", "p3", "p4", "p5")
age <- c(15, 32, 44, 25, 39)
gender <- c("F","M","M","F","F")
insure <- c(F,F,F,F,F)
```

Exercise

Create a R object from these vectors and change vector `insure` to `T` for each patient who is a female and above the age of 20

```
df <- data.frame(patientsID,age,gender,insure,stringsAsFactors = F)

for(patientInd in 1:dim(df)[1]){
  if(df[patientInd,"gender"]=="F" & df[patientInd,"age"]>20){
    df[patientInd,"insure"] = T
  }
}

df[,"insure"]
```

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

functions

functions

R does everything by copying. What do we mean by that?

```
x <- 3
y <- x
x <- 2
x
y
```

functions

R does everything by copying. What do we mean by that?

```
x <- 3
y <- x
x <- 2
x
```

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

```r
y
```

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

functions

- A `function` is a block of code that can be used multiple times. It is a way to structure the code.
- R provides many built-in functions that ease writing programs and executing codes.
- Examples: `print`, `:`, `c`, `in`

functions

R also provides users with the ability to write their own custom functions

```r
printThanks <- function(){
  print("Thank you very much for running this code")
}

printThanks()
```

```
## [1] "Thank you very much for running this code"
```

functions

```r
isEven <- function(x){
  if(x%%2==0){
    return(TRUE)
  }
  return(FALSE)
}
isEven(4)
```

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

```r
x <- isEven(3)
x
```

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

functions

Variables defined inside a function are local to that function.

```r
x <- 5
testScope <- function(x){
  return(x+1)
}
testScope(4)
testScope(x)
x
```

functions

Variables defined inside a function are local to that function.

```
x <- 5
testScope <- function(x){
  return(x+1)
}
testScope(4)
```

```
## [1] 5
```

```
testScope(x)
```

```
## [1] 6
```

```
x
```

```
## [1] 5
```

*apply

Functions in R become super useful together with the built-in apply functions. *apply helps you apply a function to each element of a vector or row/column of a matrix.

There are 3 main apply functions: `lapply`, `sapply` and `apply`. `lapply` - l stands for list, will return a list. `sapply` s stands for simple, will try to simplify the list. `apply` is `sapply` for matrices and data.frames.

sapply

Here is an sapply example:

```
isEven <- function(x){
  if(x%%2==0){
    return(TRUE)
  }
  return(FALSE)
}

myVec <- c(4,3,2,1)
sapply(myVec, isEven)
```

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

apply

`apply`, unlike sapply and lapply, takes 3 arguments: the data to apply over, which dimensions to apply over, and the function to apply.

Heres a simple example, using the built-in `sum` function, which returns the sum of a vector:

```
my.mat <- matrix(1:4, nrow=2, ncol=2)
my.mat
apply(my.mat, 1, sum)
apply(my.mat, 2, sum)
```

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

functions

R has many built-in functions that can ease writing programs and you can find their syntax by using `?` operator

Examples:

```
?mean
?rownames
```

```
## Help on topic 'rownames' was found in the following packages:
##
##   Package               Library
##   DBI                   /home/arvind/.R
##   tibble                /home/arvind/.R
##   base                  /usr/lib/R/library
##
##
## Using the first match ...
```

```
?read.csv
```

Exercise

Write a function will return T if a given element exists in a given vector.

Exercise

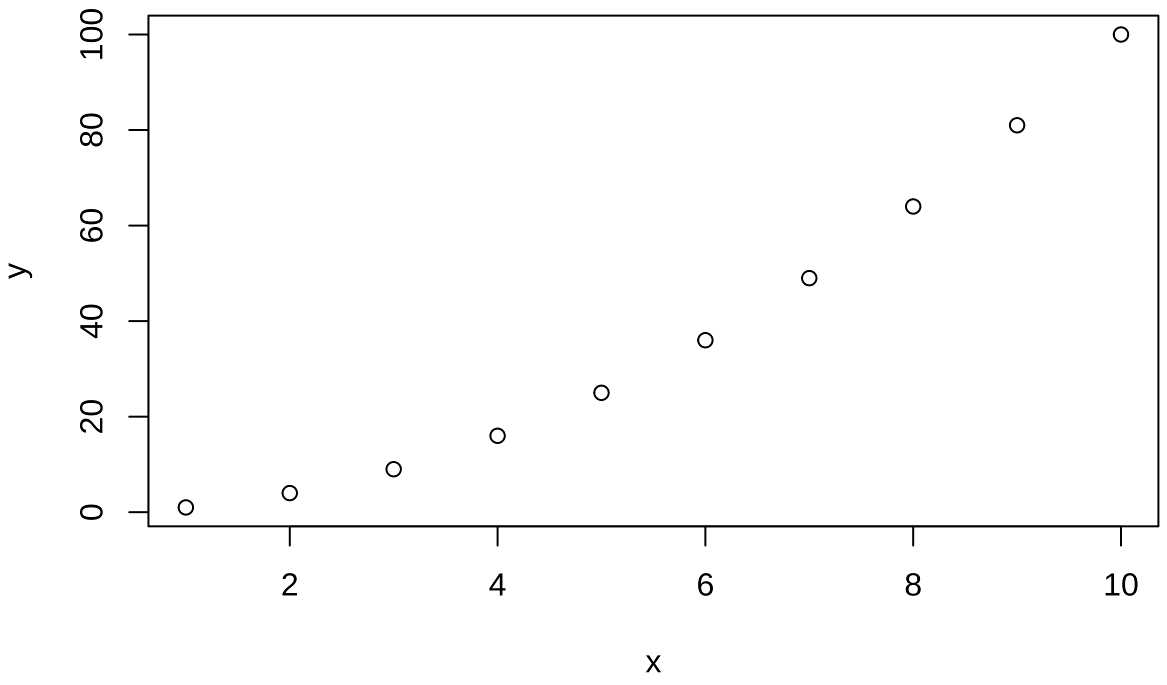
```
exists <- function(vector, element){

  for (v1 in vector) {
    if(v1 == element){
      return(TRUE)
    }
  }
  return(FALSE)
}
```

Plots

Plots

Plotting in R is easy:

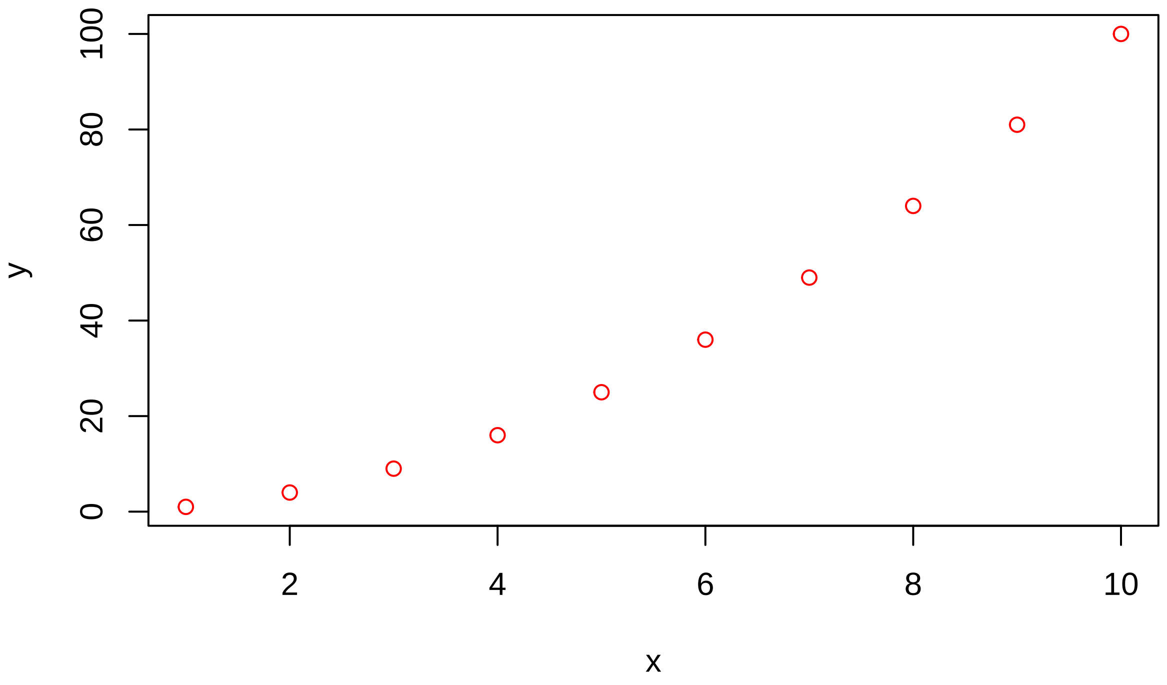
```
x = 1:10
y = x**2
plot(x, y)
```



Plot

Lets change color

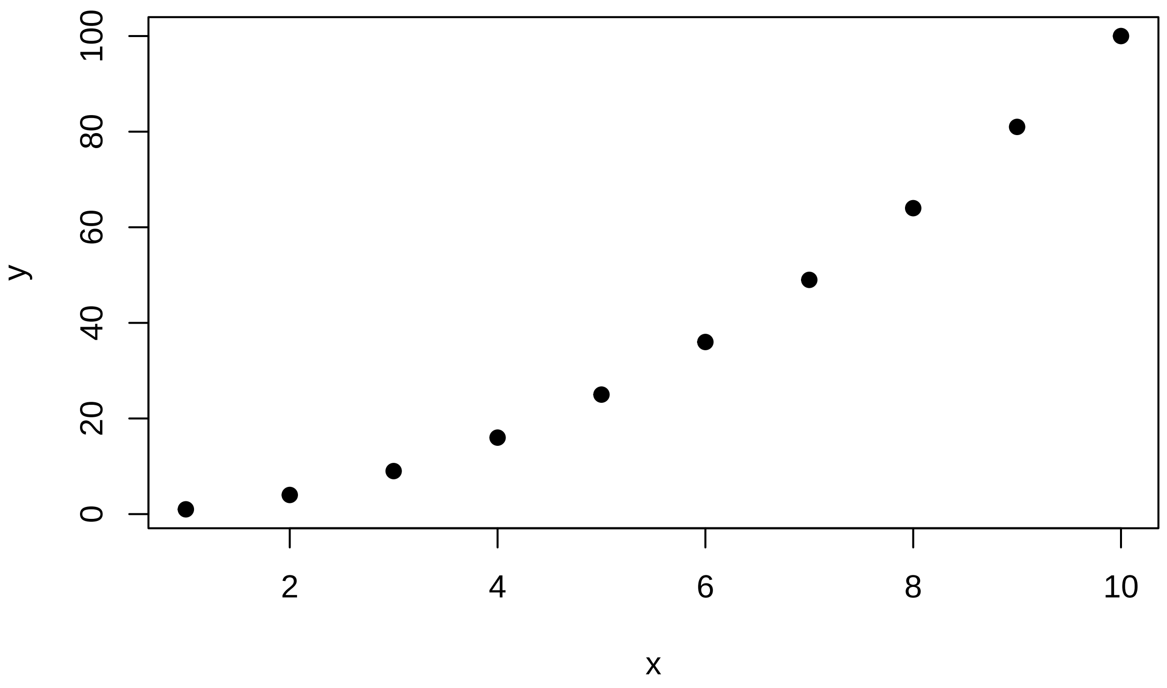
```
plot(x, y, col = "red")
```



Plot

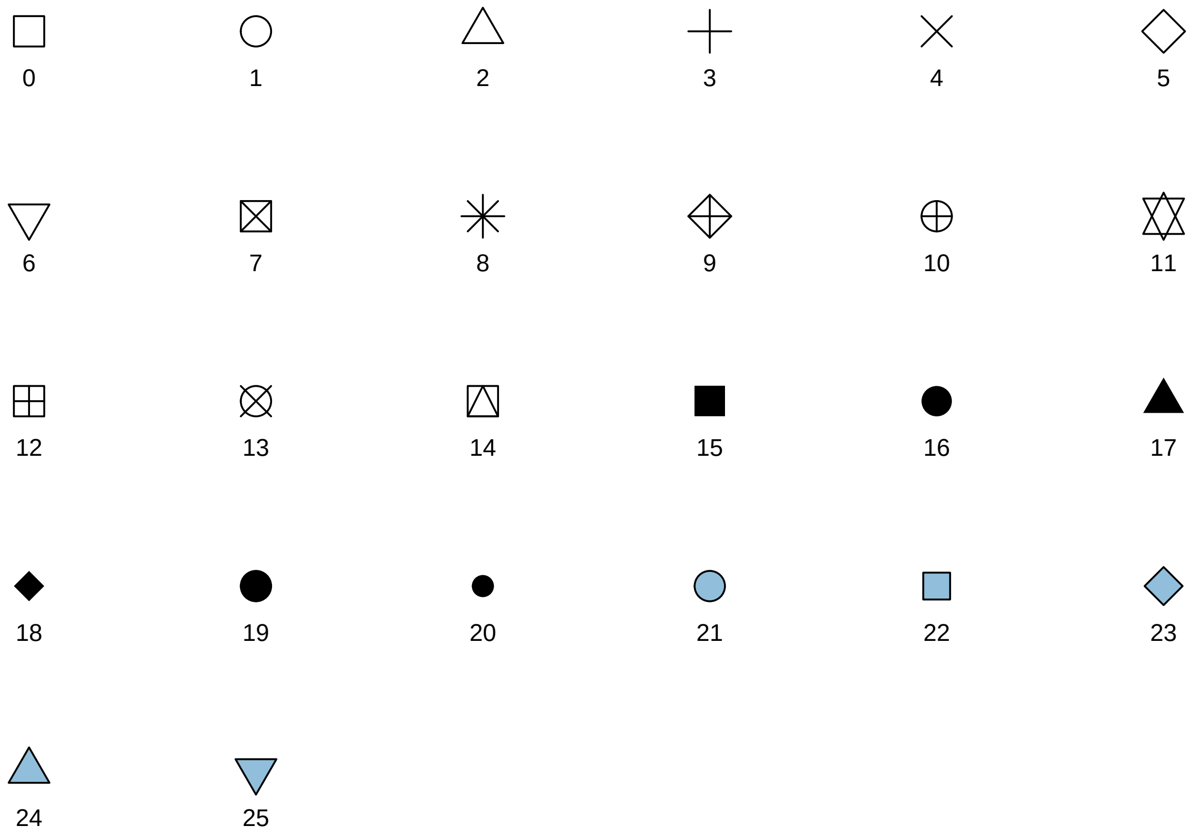
Lets change point type

```r
plot(x, y, pch = 19)
```



PCH

PCH (plotting character) is an argument to specify point shapes



PCH

pch = 0, square

pch = 13, circle cross

pch = 1, circle

pch = 14, square and triangle down

pch = 2, triangle point up

pch = 15, filled square

pch = 3, plus

pch = 16, filled circle

pch = 4, cross

pch = 17, filled triangle point-up

pch = 5, diamond

pch = 18, filled diamond

pch = 6, triangle point down

pch = 19, solid circle

pch = 7, square cross

pch = 20, bullet (smaller circle)

pch = 8, star

pch = 21, filled circle blue

pch = 9, diamond plus

pch = 22, filled square blue

pch = 10, circle plus

pch = 23, filled diamond blue

pch = 11, triangles up and down

pch = 24, filled triangle point-up blue

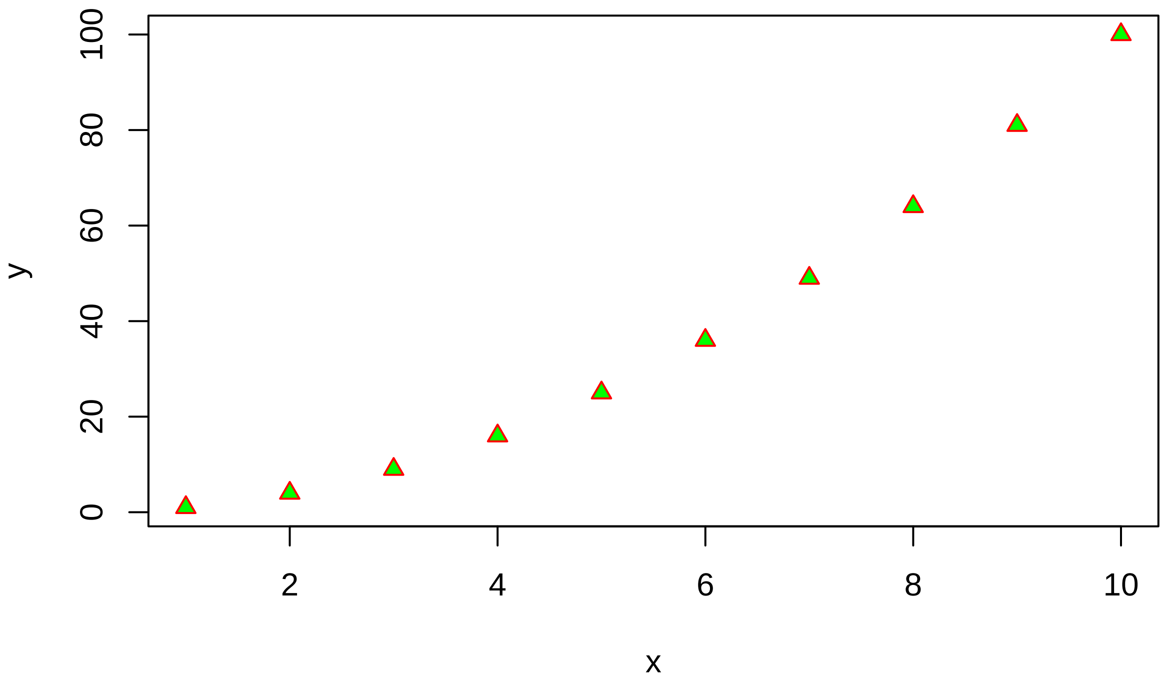
pch = 12, square plus

pch = 25, filled triangle point down blue

Plot

- For PCH 21 to 25, background color can be specified

```
plot(x, y, pch = 24, col="red", bg="green")
```



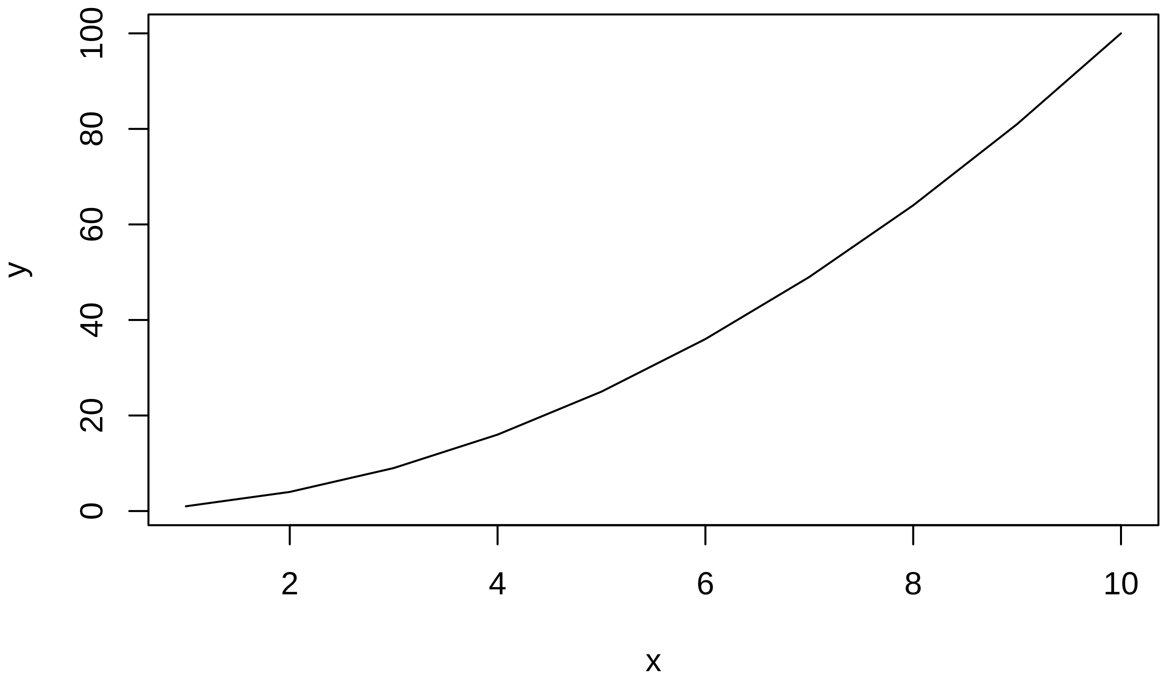
Task

- plot data points in your favorite color and shape

Plot Line

The option `"type"` can change the plot type

```
plot(x, y, type = "l")
```



Plot type

“p”

for points

“l”

for lines

“b”

for both

“c”

for the lines part alone of “b”

“o”

for both ‘overplotted’

“h”

for ‘histogram’ like (or ‘high-density’) vertical lines

“s”

for stair steps

“S”

for other steps

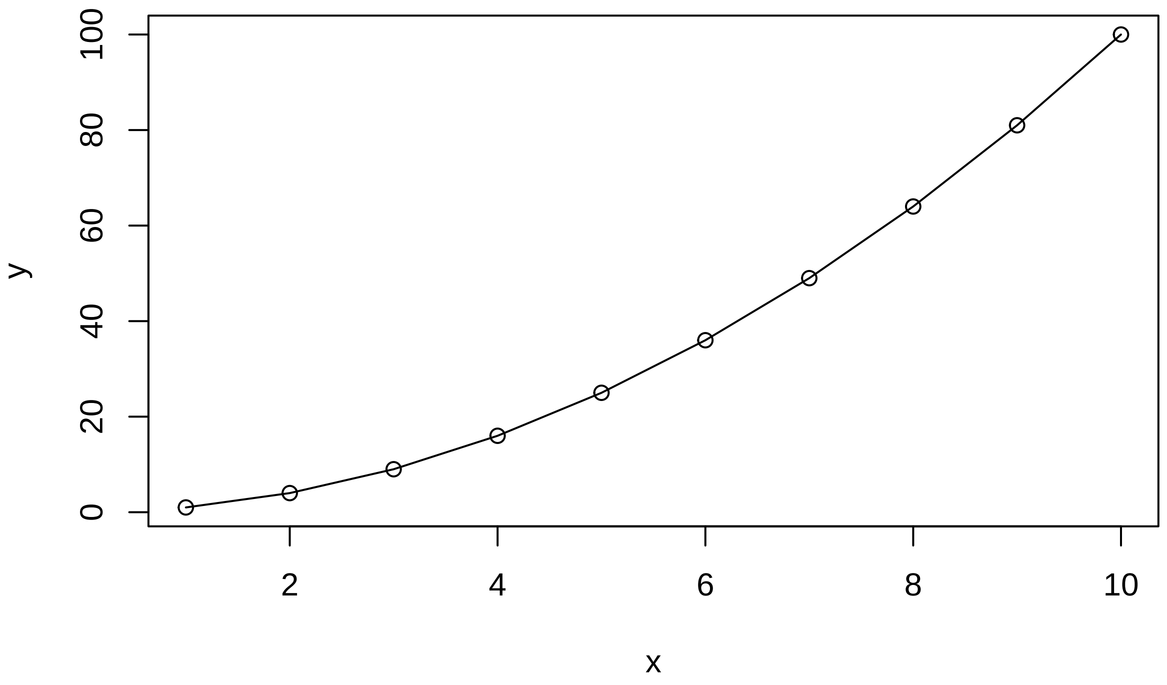
“n”

for no plotting

Plot

Lets plot both point and line

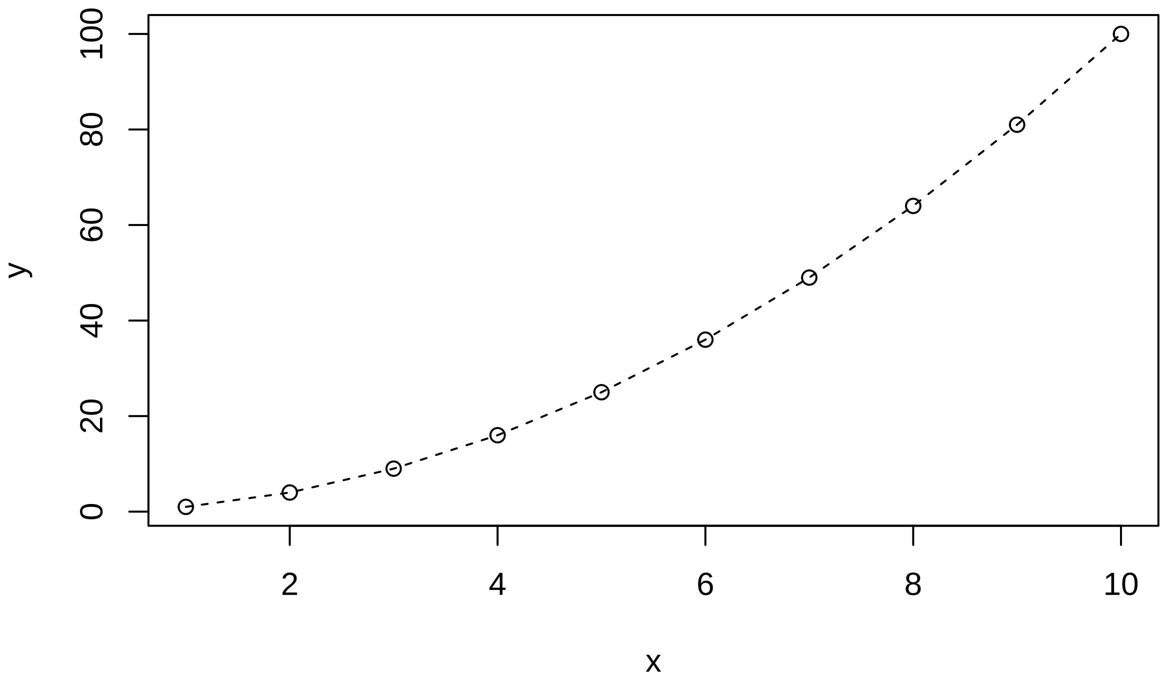
```
plot(x, y, type = "o")
```



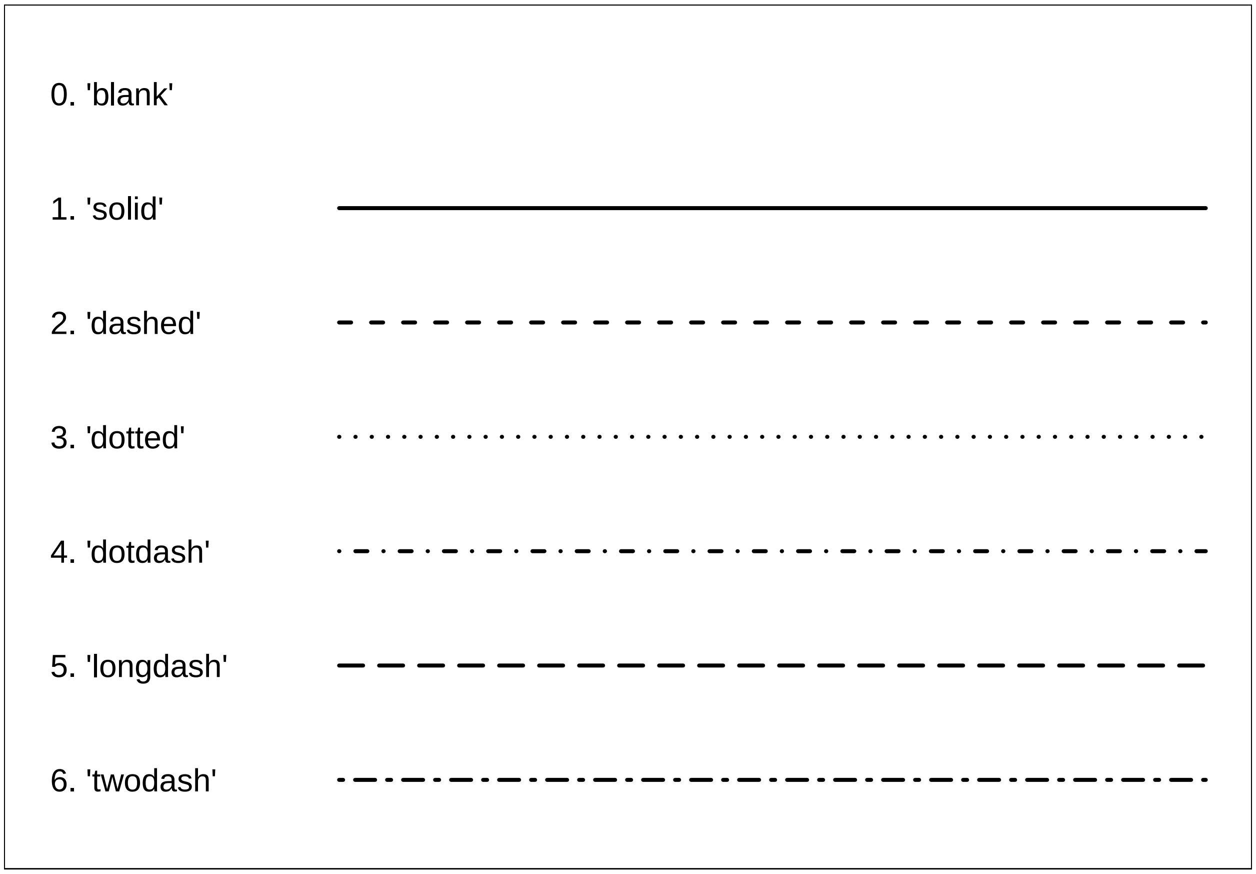
Change line type

The option "lty" can change the line type

```
plot(x, y, type = "o", lty=2)
```



Line types in R



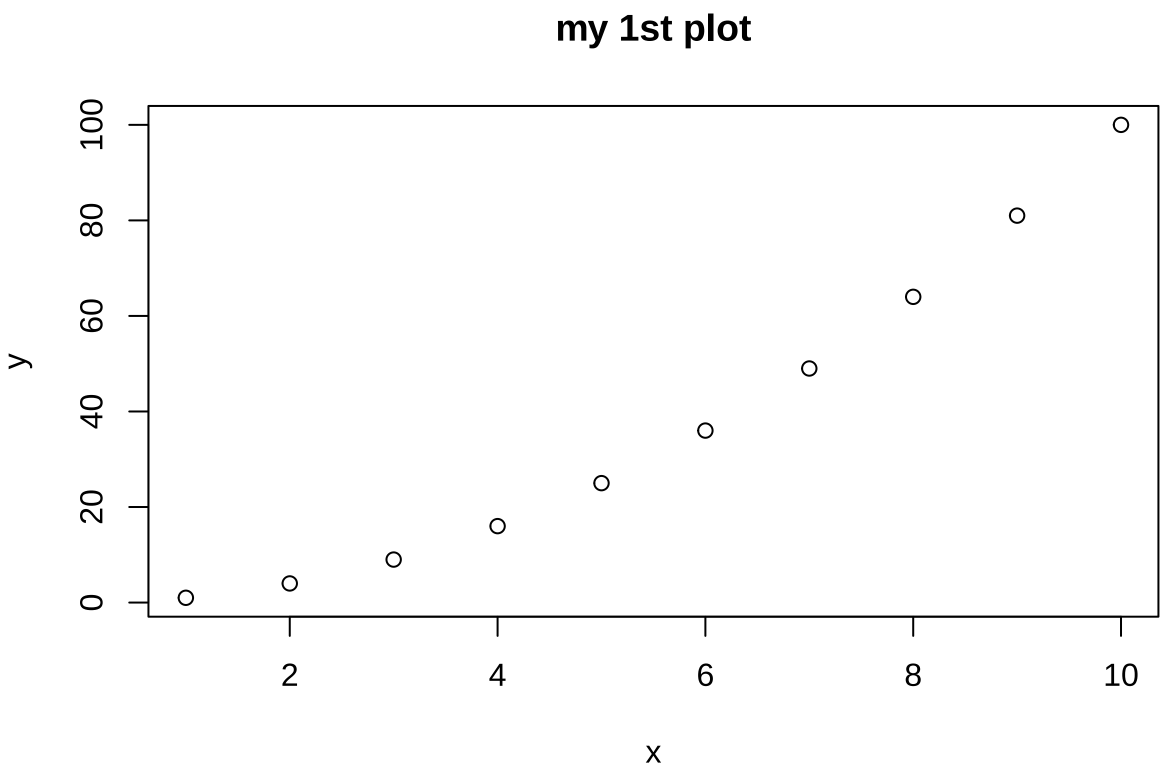
Task

- plot data points and line in your favorite color and shape

Plot Title

Give plot a title

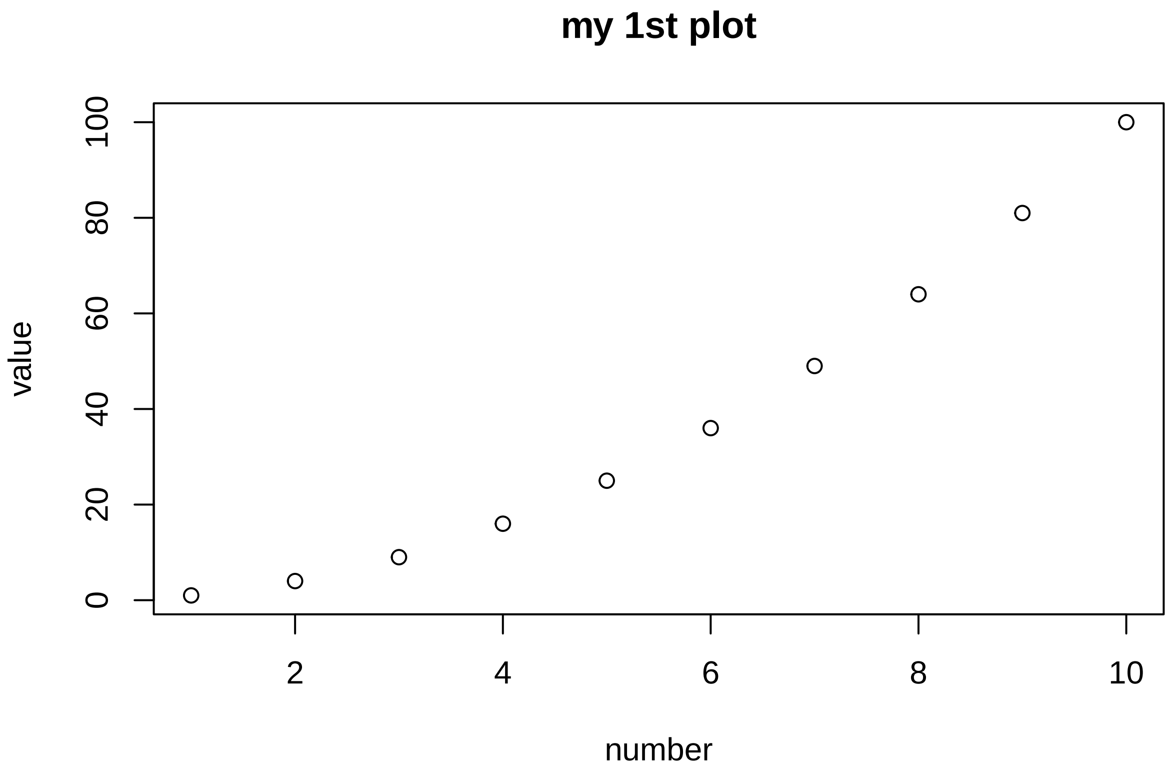
```
plot(x, y, main = "my 1st plot")
```



Axis Name

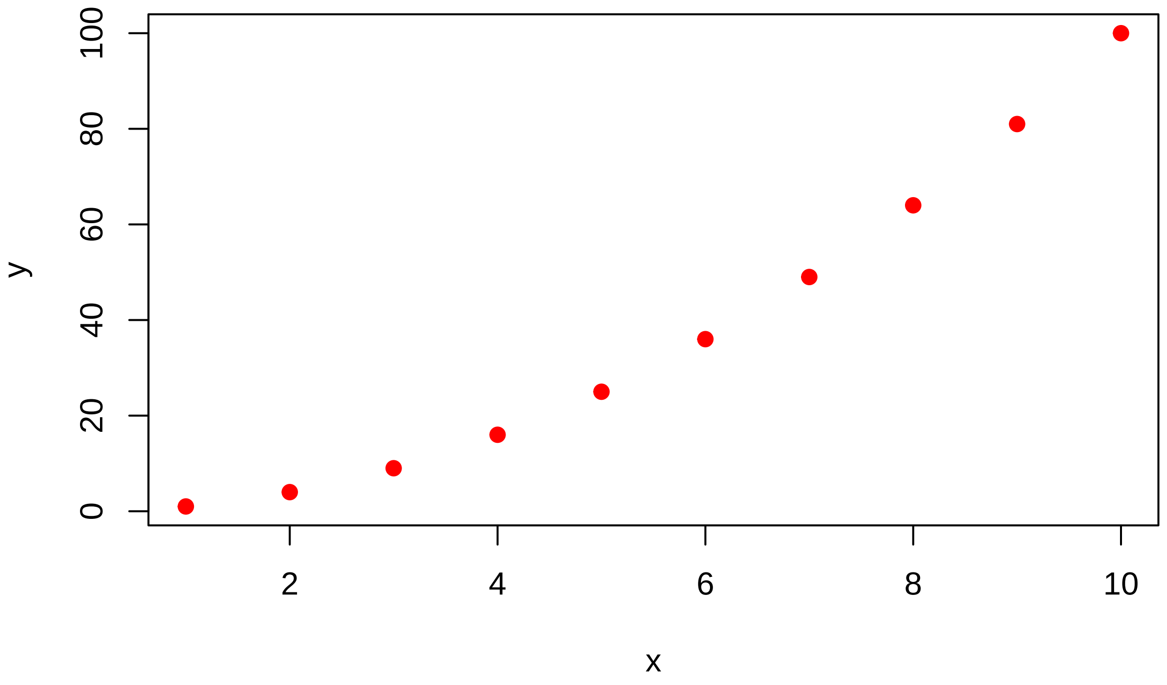
Change plot axis name

```
plot(x, y, main = "my 1st plot", xlab = "number", ylab = "value")
```



Adding more to a plot

```
x = 1:10
y = x**2
z = y + 1
plot(x, y, pch=19, col="red")
```



Adding more to a plot

`{r}, fig.height = 4.0,  fig.align = "center"} x = 1:10  y = x**2  z = y + 10 plot(x, y, pch=19, col="red") lines(x, y, col="red") points(x, z, col="blue") lines(x, z, col="blue")`

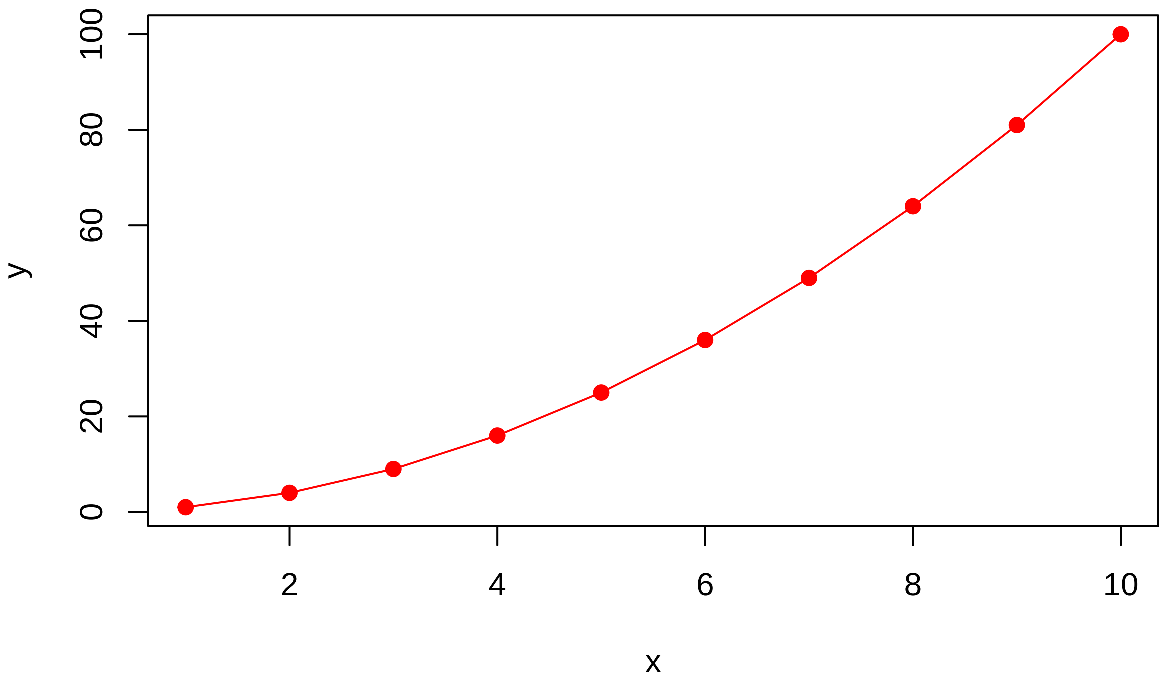
Task

- plot data points and line in your favorite color and shape

Fixing axis limits

- In R plots are static
- Adding new points can't change the axis limits

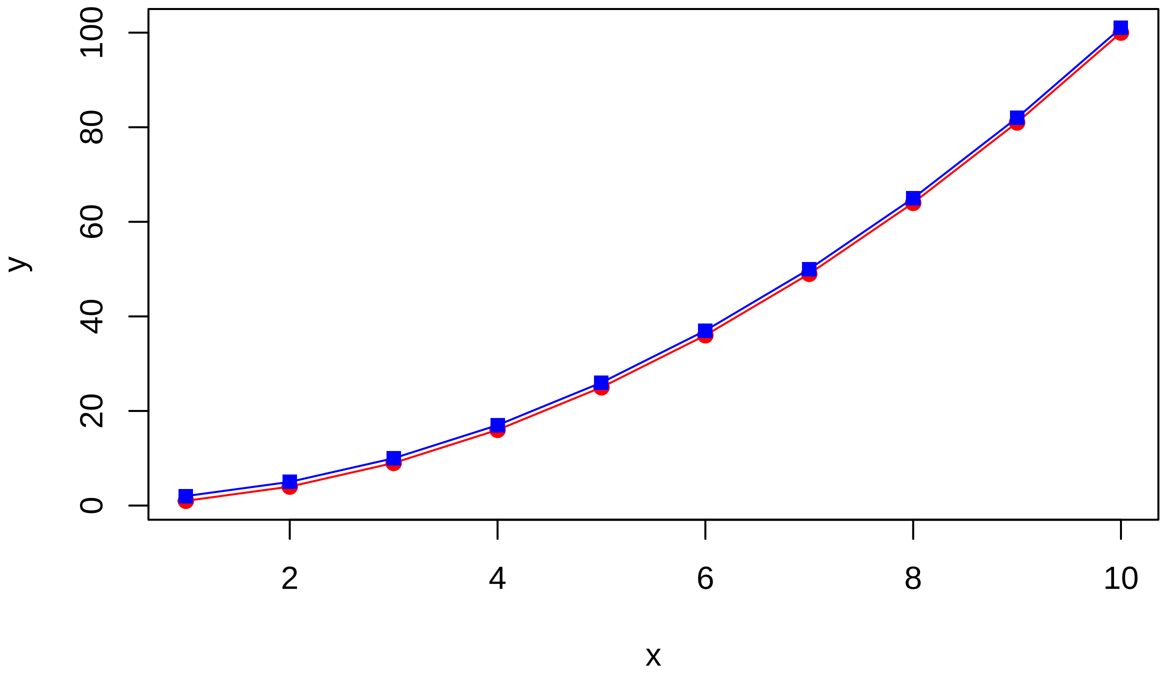
```
plot(x, y, pch=19, col="red")
lines(x, y, col="red")
```



Fixing axis limits

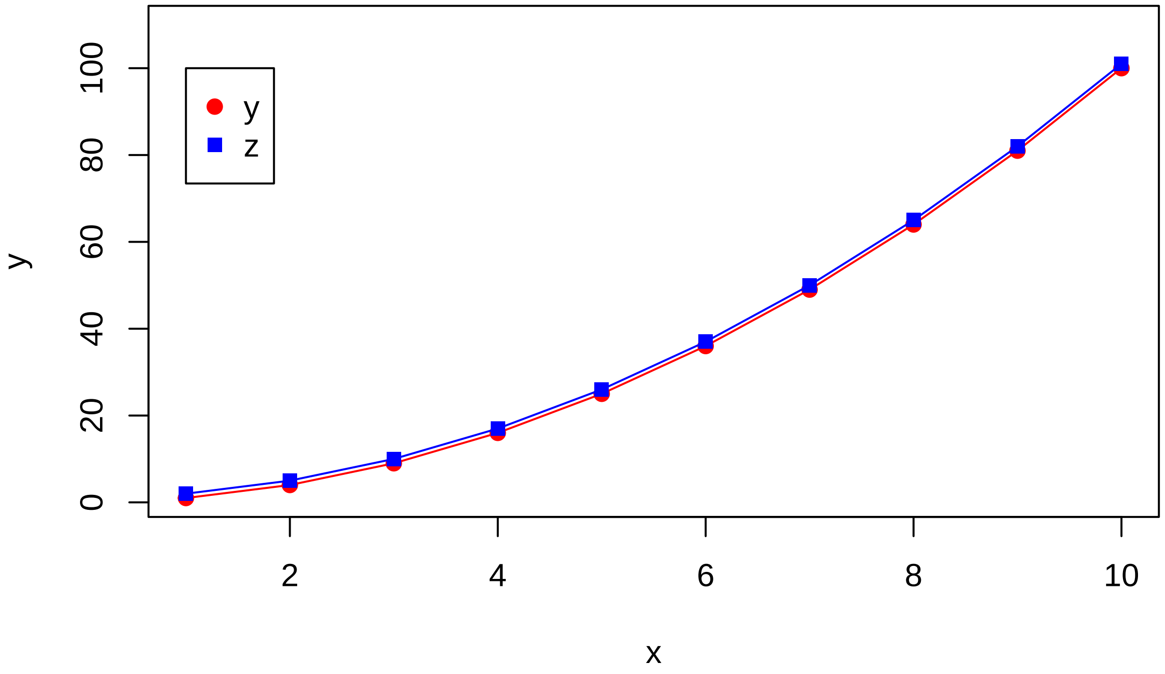
To overcome this issue specify the axis limits in advance

```
ylimit = range(c(y,z))
plot(x, y, col="red", pch=19, ylim = ylimit)
lines(x, y, col="red")
points(x, z, col="blue", pch=15)
lines(x, z, col="blue")
```



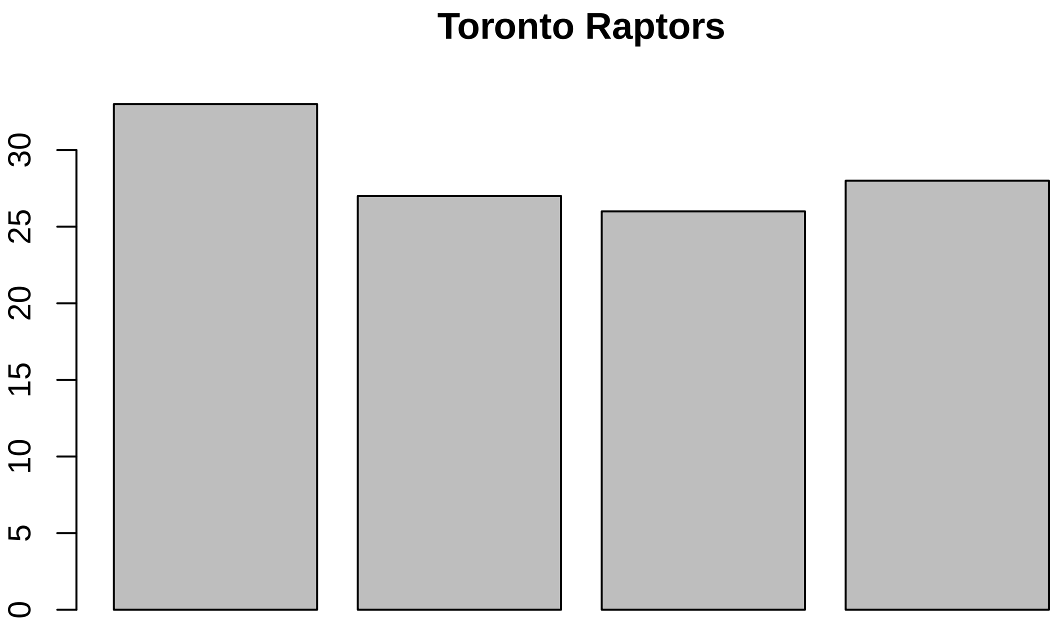
Adding legend

```
plot(x, y, col="red", pch=19, ylim = c(1,110))
lines(x, y, col="red")
points(x, z, col="blue", pch=15)
lines(x, z, col="blue")
legend(1,100,legend=c("y","z"), col=c("red","blue"), pch=c(19,15))
```



Barplot

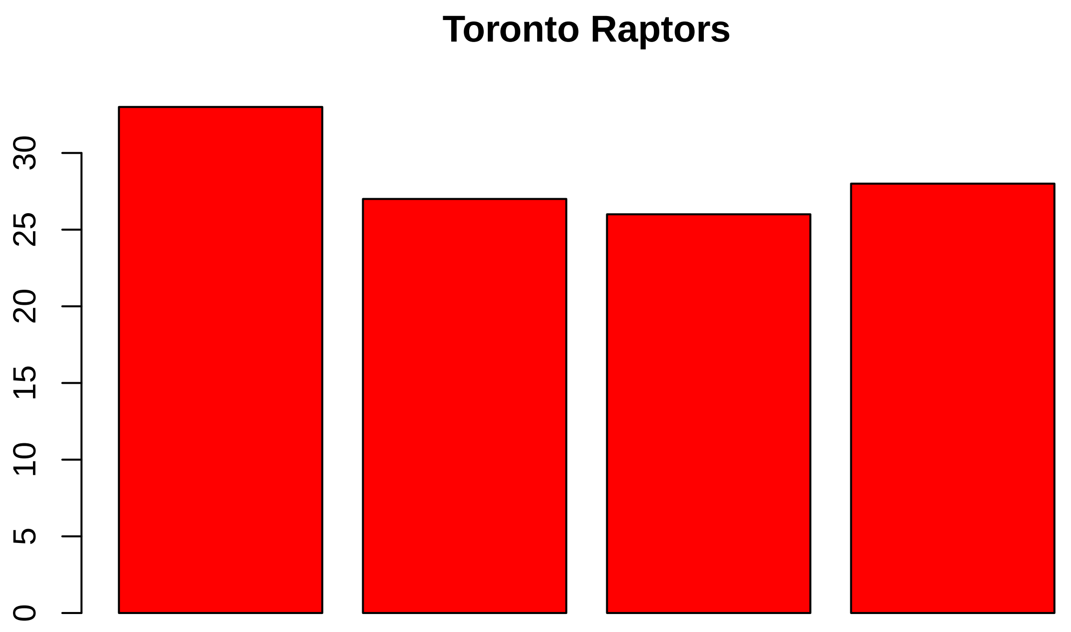
```
raptors = c(33,27,26,28)
barplot(raptors, main="Toronto Raptors")
```



How do we change the bar color?

Barplot

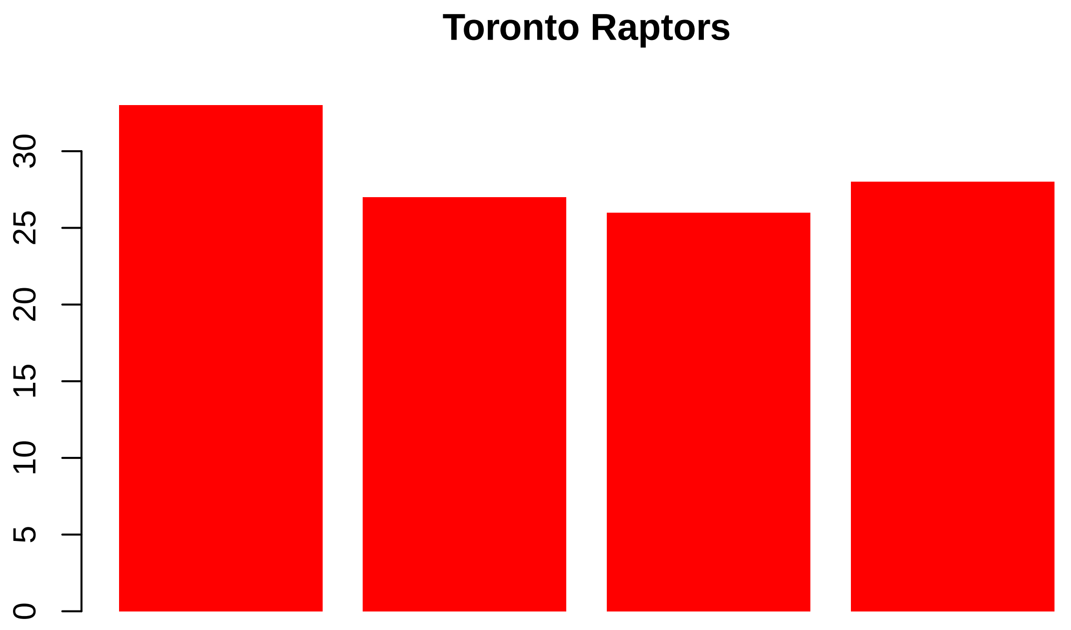
```r
raptors = c(33,27,26,28)
barplot(raptors, main="Toronto Raptors", col="red")
```



How do we change border color?

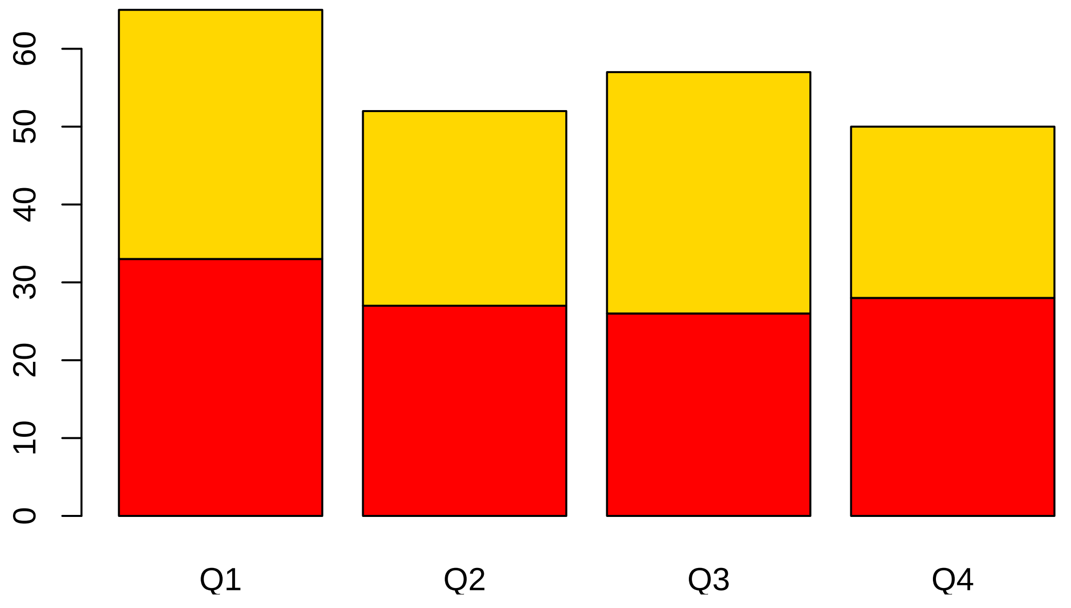
Barplot

```r
raptors = c(33,27,26,28)
barplot(raptors, main="Toronto Raptors", col="red", border = NA)
```



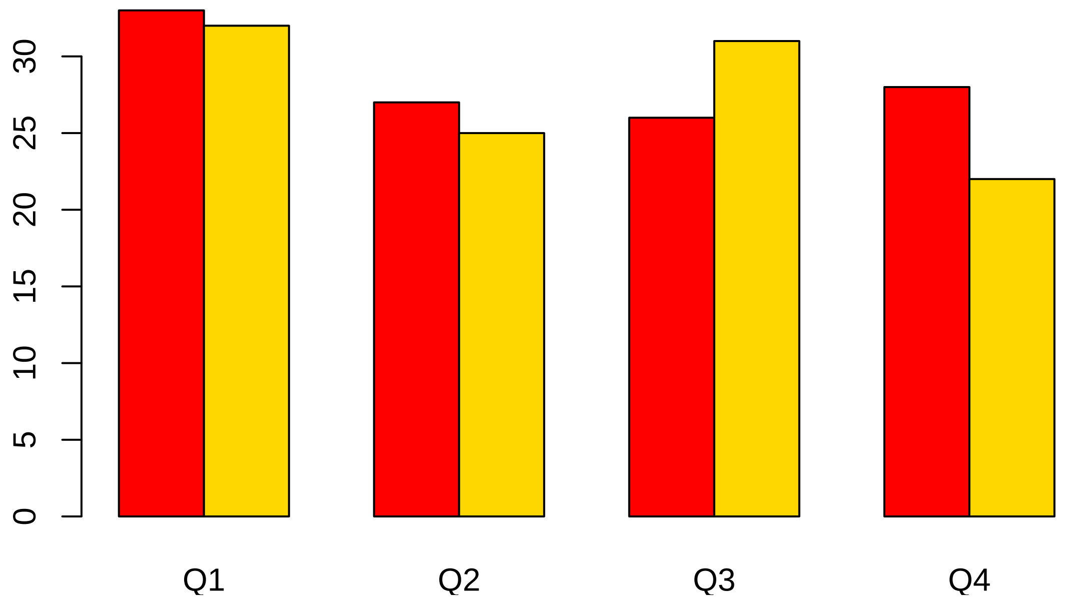
Barplot (Stacked)

```
raptors = c(33,27,26,28)
gsw = c(32, 25, 31, 22)
data = rbind(raptors, gsw)
colnames(data) = c("Q1", "Q2", "Q3", "Q4")
barplot(data, col=c("red", "gold"))
```



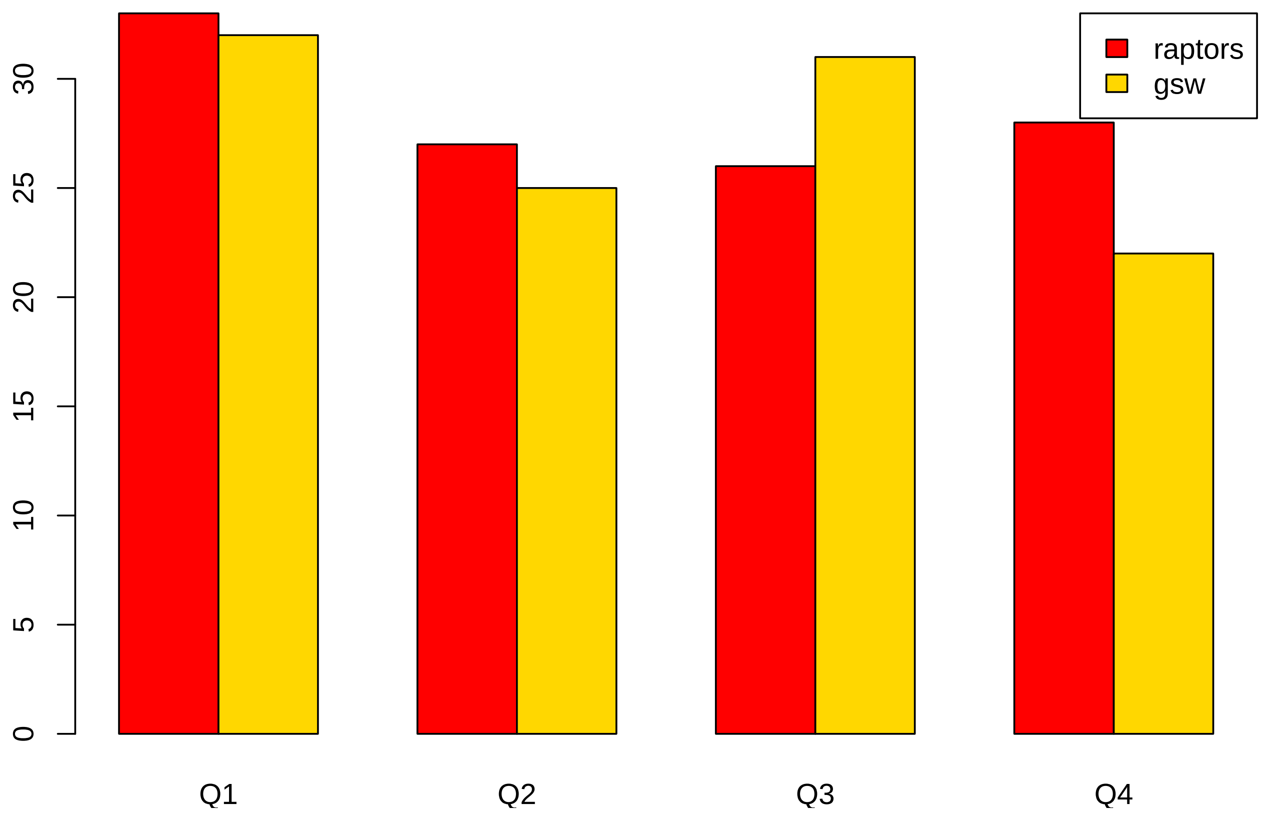
Barplot (Grouped)

```
barplot(data, col=c("red", "gold"), beside=TRUE)
```



Barplot (add legend)

```
barplot(data, col=c("red","gold"), beside=TRUE, legend=rownames(data),
        args.legend = list(x="topright"))
```

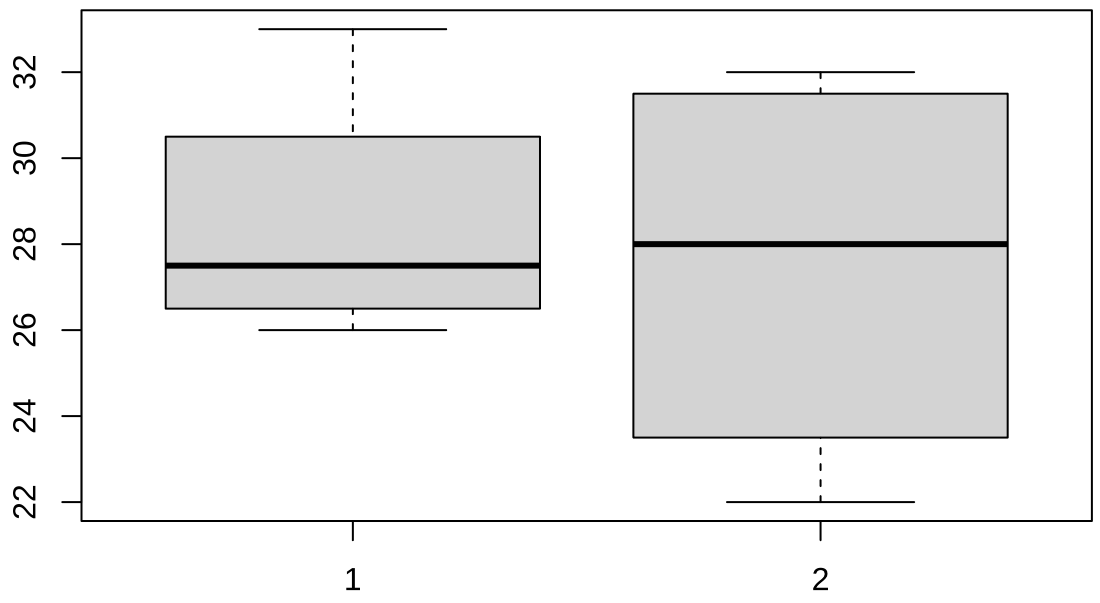


Boxplot

Boxplot

Same as barplot but data should be in column:

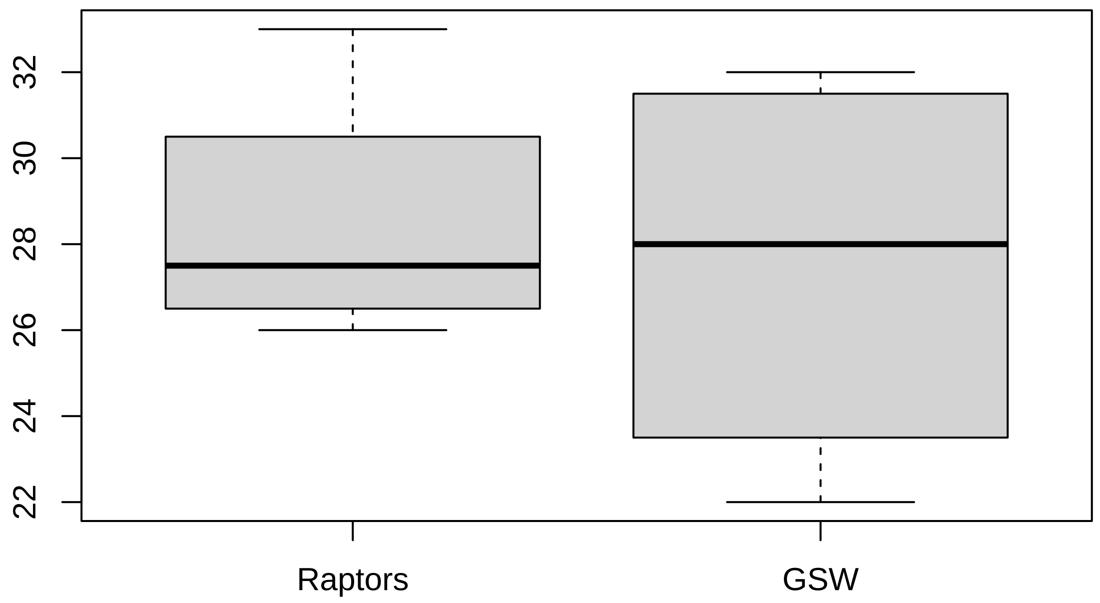
```
raptors = c(33,27,26,28)
gsw = c(32, 25, 31, 22)
boxplot(raptors, gsw)
```



Boxplot

Add names to the box:

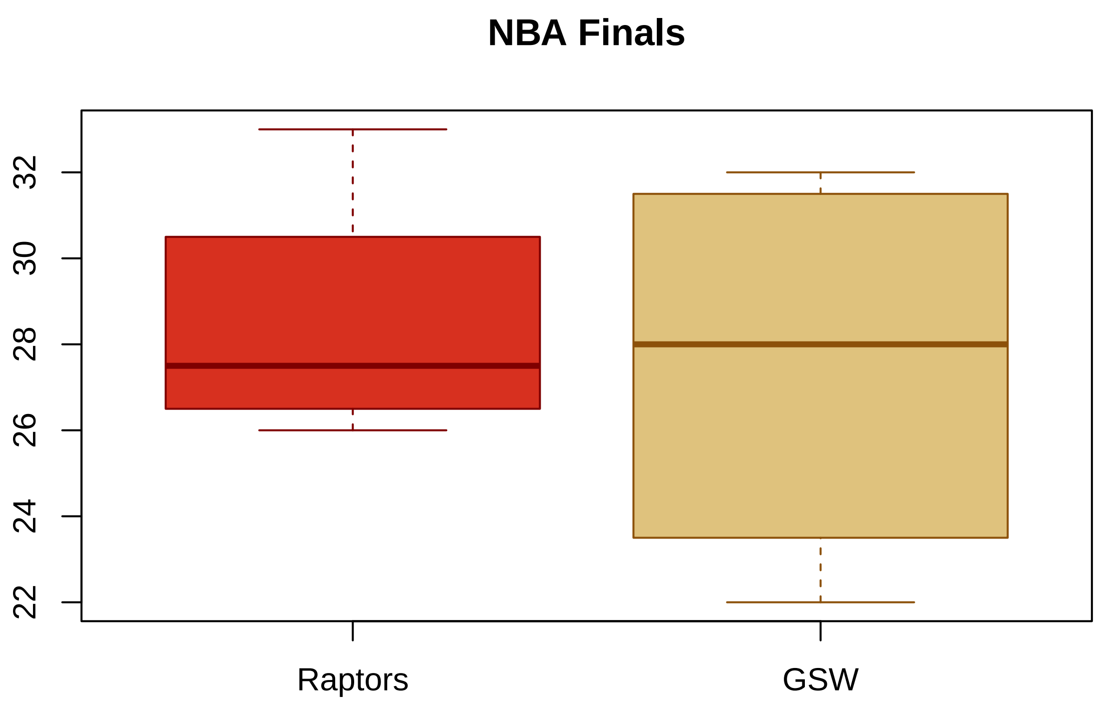
```
raptors = c(33,27,26,28)
gsw = c(32, 25, 31, 22)
boxplot(raptors, gsw, names=c("Raptors","GSW"))
```



Boxplot (color)

Here HTML color codes are used:

```
boxplot(raptors, gsw, names=c("Raptors", "GSW"),
        col=c("#d7301f","#dfc27d"),
        border=c("#7f0000", "#8c510a"), main="NBA Finals")
```



Task

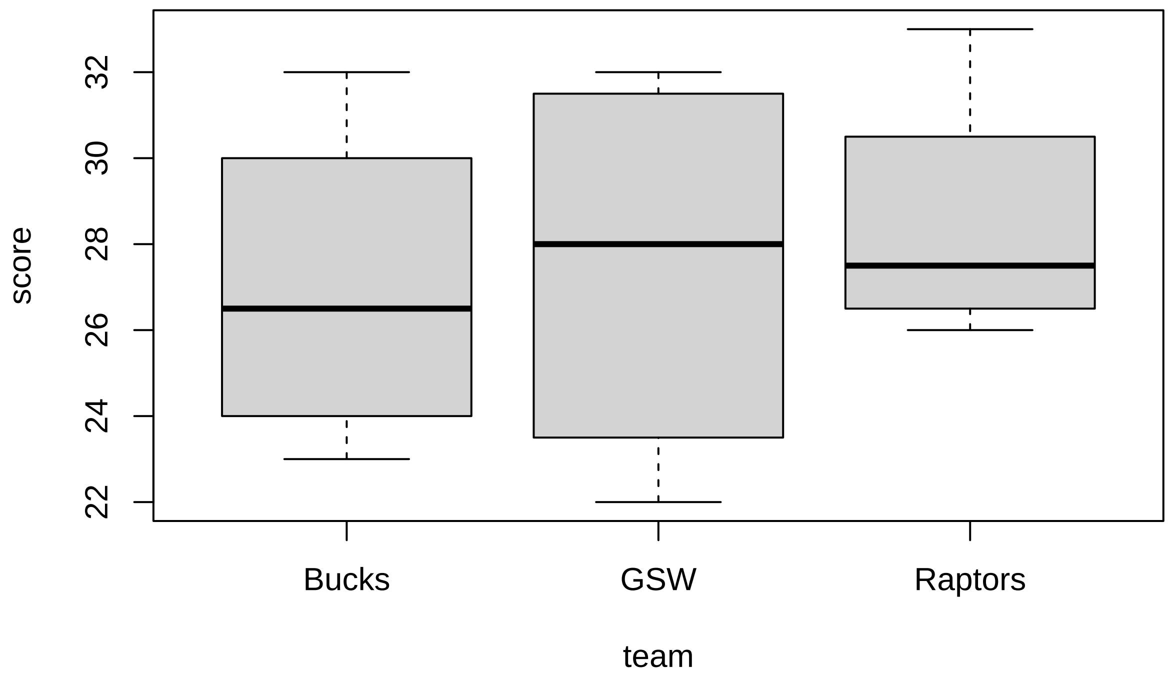
- Add one more boxplot with data:

```
bucks = c(23, 28, 25, 32)
```

Using formula

Here HTML color codes are used:

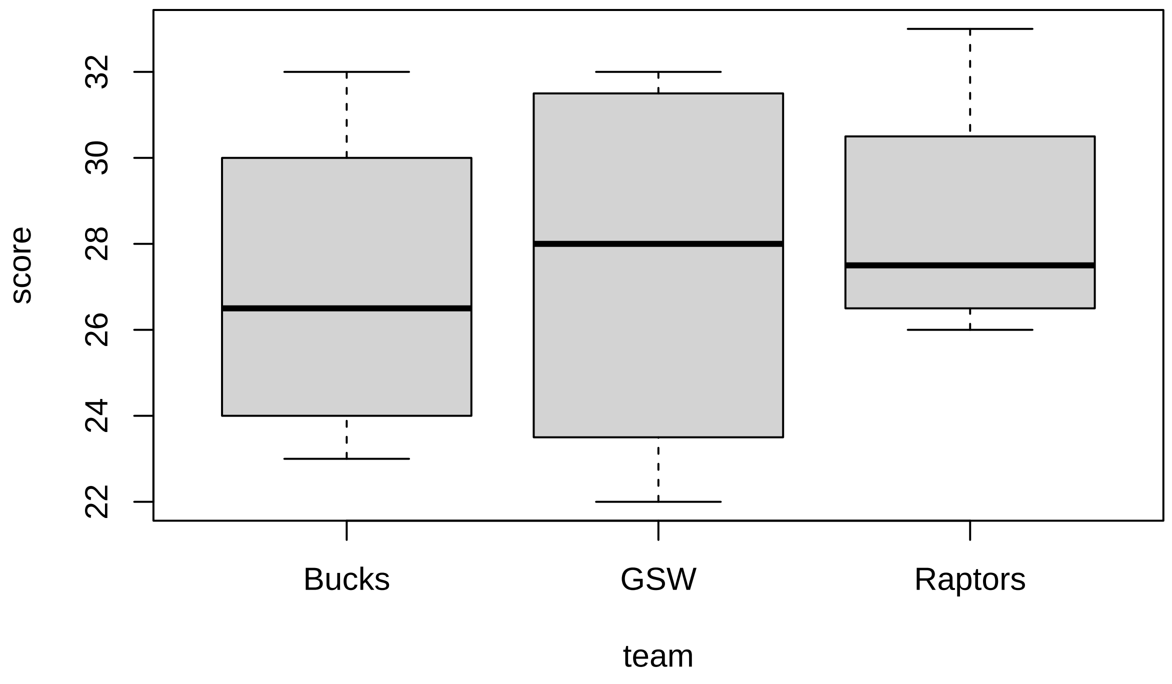
```
bucks = c(23, 28, 25, 32)
df=data.frame(score=c(raptors, gsw, bucks),
              team= c(rep("Raptors", 4), rep("GSW", 4), rep("Bucks", 4)))
boxplot(score~team, data=df)
```



Using formula

Here HTML color codes are used:

```
bucks = c(23, 28, 25, 32)
df=data.frame(score=c(raptors, gsw, bucks),
              team=c(rep("Raptors",4), rep("GSW",4), rep("Bucks",4)))
boxplot(score~team, data=df)
```



Task

- Add one more boxplot with data:

Saving Plots for Publication

- For publications use PDF or PNG format
- To save plot, open file just before plot function
- Don't forget to close the file using **dev.off()**

```
raptors = c(33,27,26,28)
gsw = c(32, 25, 31, 22)
bucks = c(23, 28, 25, 32)

pdf("NBA_finals.pdf", width = 4, height = 5)

boxplot(raptors, gsw, bucks, names=c("Raptors", "GSW", "Bucks"),
        col=c("#d7301f","#dfc27d", "#41ab5d"),
        border=c("#7f0000", "#8c510a", "#00441b"), main="NBA Finals")

dev.off()
```

Saving Plots for Publication

- In **PNG** format, specify resolution using **res**
- **res** should be at least 300 for publications and 600 for presentation
- Also, don't forget to specify **units**

```
png("figure-1.png", width = 4, height = 5, units = 'in', res = 600)

boxplot(raptors, gsw, bucks, names=c("Raptors", "GSW", "Bucks"),
        col=c("#d7301f","#dfc27d", "#41ab5d"),
        border=c("#7f0000", "#8c510a", "#00441b"), main="NBA Finals")

dev.off()
```

R data

- R comes with several inbuilt datasets

```
data("iris")
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
```

packages

Packages

- R comes with basic functions that you will use on a daily basis

- Sometime you will require specialized functions, written by others
- R package is a collection of functions and data written for a specific task

Install a package

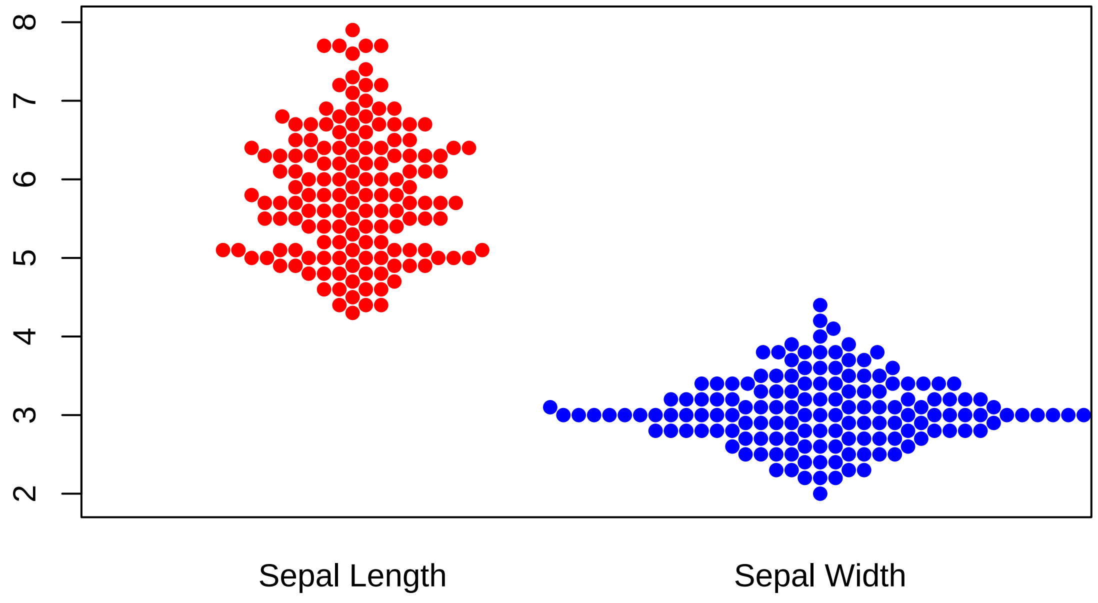
- To install an R package use the command **install.packages("package name")**
- In this example, we install the package **beeswarm**

```
install.packages("beeswarm")
```

Load a package

- Now we load the package and use a function from it

```
library("beeswarm")
data("iris")
beeswarm(iris[, c("Sepal.Length", "Sepal.Width")],
         pch = 16, col=c("red", "blue"),
         labels = c("Sepal Length", "Sepal Width"))
```



Finding Help

- All functions in R are documented
- **?** can be used to find documents

```
?beeswarm
```

- **??** can be used to find something anywhere in R documents

```
??beeswarm
```

Now find more about the function mean

Finding R package

- Google the problem
- Look into CRAN website www.cran.r-project.org/web/packages/

- Look into Bioconductor repo www.bioconductor.org
- ** Note:** packages at Bioconductor uses a different command to install that is:

```
BiocManager::install("package name")
```

reading and writing files

Save data

- R can save data in different formats
- Data can be saved in R native format

```
raptors = c(33,27,26,28)
gsw = c(32, 25, 31, 22)
df = cbind(raptors, gsw)
saveRDS(df, "my_data.rds")
```

- To read it use readRDS function

```
mydata = readRDS("my_data.rds")
head(mydata)
```

```
##      raptors gsw
## [1,]      33  32
## [2,]      27  25
## [3,]      26  31
## [4,]      28  22
```

Save data

- R can save data in CSV formats

```
write.csv(df,file="my_data.csv")
```

- To read it use read.csv function

```
mydata2 = read.csv("my_data.csv", row.names = 1)
head(mydata2)
```

```
##   raptors gsw
## 1      33  32
## 2      27  25
## 3      26  31
## 4      28  22
```

Excel files

- R package **openxlsx** can read and write Excel files
- Task: Install the package **openxlsx** and load it

Write excel files

- To install the package

```
install.packages("openxlsx")
```

- To write the Excel file

```
library(openxlsx)
write.xlsx(df, "myData.xlsx", overwrite = TRUE)
```

- Now open the **myData.xlsx** file, add one more row, save and close the file

Read excel files

```
myxl = read.xlsx("myData.xlsx")
head(myxl)
```

```
## [1] 22
## <0 rows> (or 0-length row.names)
```

Caution

- Large Excel files may take a long time and memory to read/write
- Use R native **RDS** files to read and write the data during the analysis
- To share data, prefer **CSV** format then Excel
- Save only final results/data in Excel format

R Biostat. and data analysis

R Biostat. basics

Useful functions:

- sum(x) –> Sums the elements in x
- prod(x) –> Product of the elements in x
- max(x) –> Maximum element in x
- min(x) –> Minimum element in x
- range(x) –> Range (min to max) of elements in x

R Biostat. basics

Useful functions:

- mean(x) –> Mean (average value) of elements in x.
- median(x) –> Median (middle value) of elements in x
- var(x) –> Variance of elements in x
- sd(x) –> Standard deviation of element in x
- cor(x,y) –> Correlation between x and y
- cov(x,y) –> Covariance between x and y
- quantile(x,p) –> The pth quantile of x

R Biostat. basics

Example: patients data

```
set.seed(21341)
patients.df <- data.frame(
  "ID"=paste0("p",1:100)
  ,"Gender"=factor(sample(c("F","M"),100,replace = T))
  ,"Stage"=factor(sample(c("I","II","III","IV"),100,replace = T))
  ,"Age"=as.integer(rgamma(100,shape = 50))
  ,"TumorVolume"=rgamma(100,shape = 10)
  ,stringsAsFactors = F)

head(patients.df)
```

```
##   ID Gender Stage Age TumorVolume
## 1 p1      M    IV  43    9.255460
## 2 p2      M     I  56   13.032624
## 3 p3      F   III  59   15.205465
## 4 p4      F    II  48   10.720874
## 5 p5      M    IV  57    8.929090
## 6 p6      M   III  34    8.440177
```

R Biostat. basics

```
quantile(patients.df$Age)
```

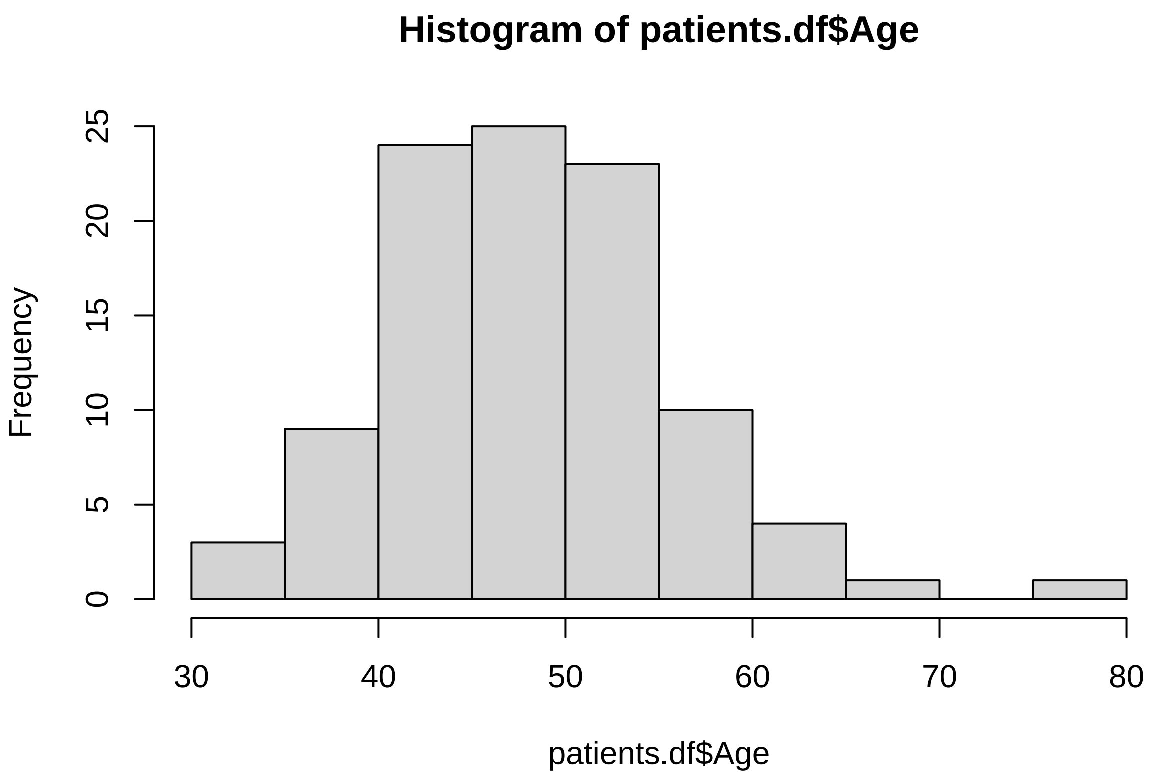
```
##   0%  25%  50%  75% 100%
##   33   43   49   53   77
```

```
summary(patients.df$Age)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   33.00   43.00   49.00   48.67   53.00   77.00
```

R Biostat. basics

```
hist(patients.df$Age)
```



R Biostat. basics

```r
summary(patients.df$Stage)
```

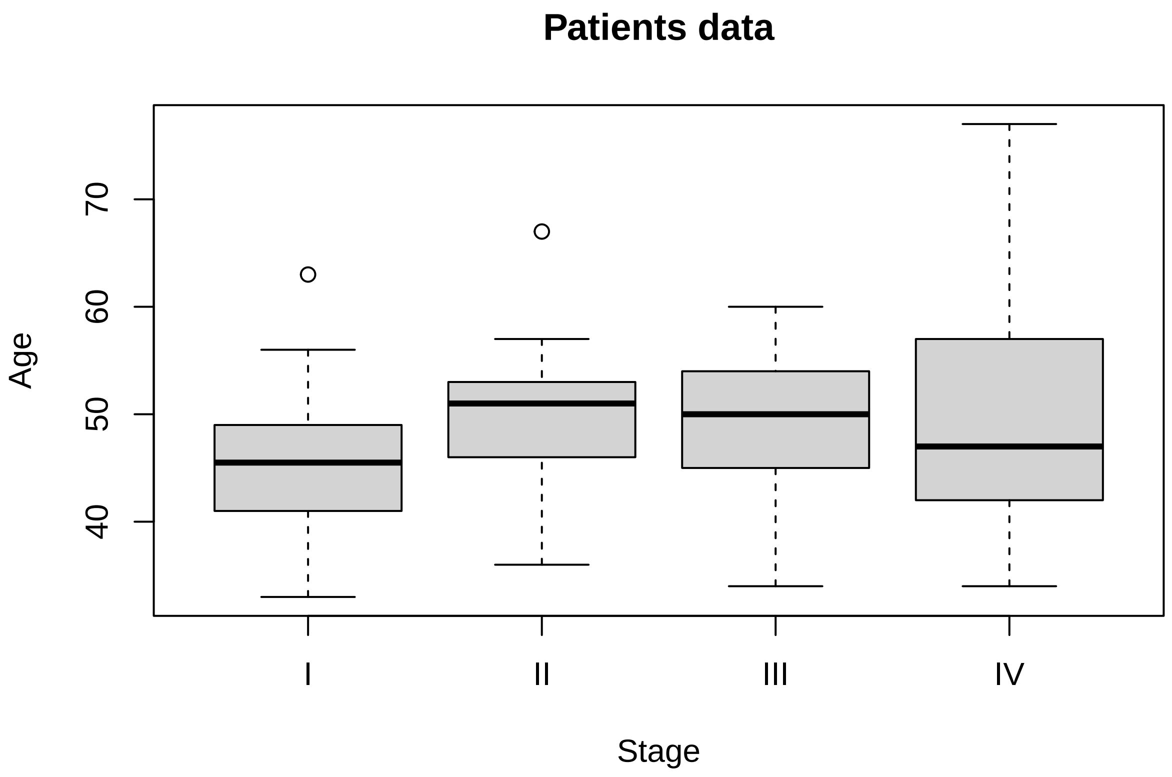
```
##   I  II III  IV
##  22  23  30  25
```

```r
table(patients.df$Stage)
```

```
##
##   I  II III  IV
##  22  23  30  25
```

R Biostat. basics

```r
boxplot(Age ~ Stage, patients.df
  , ylab="Age",xlab="Stage",main="Patients data")
```



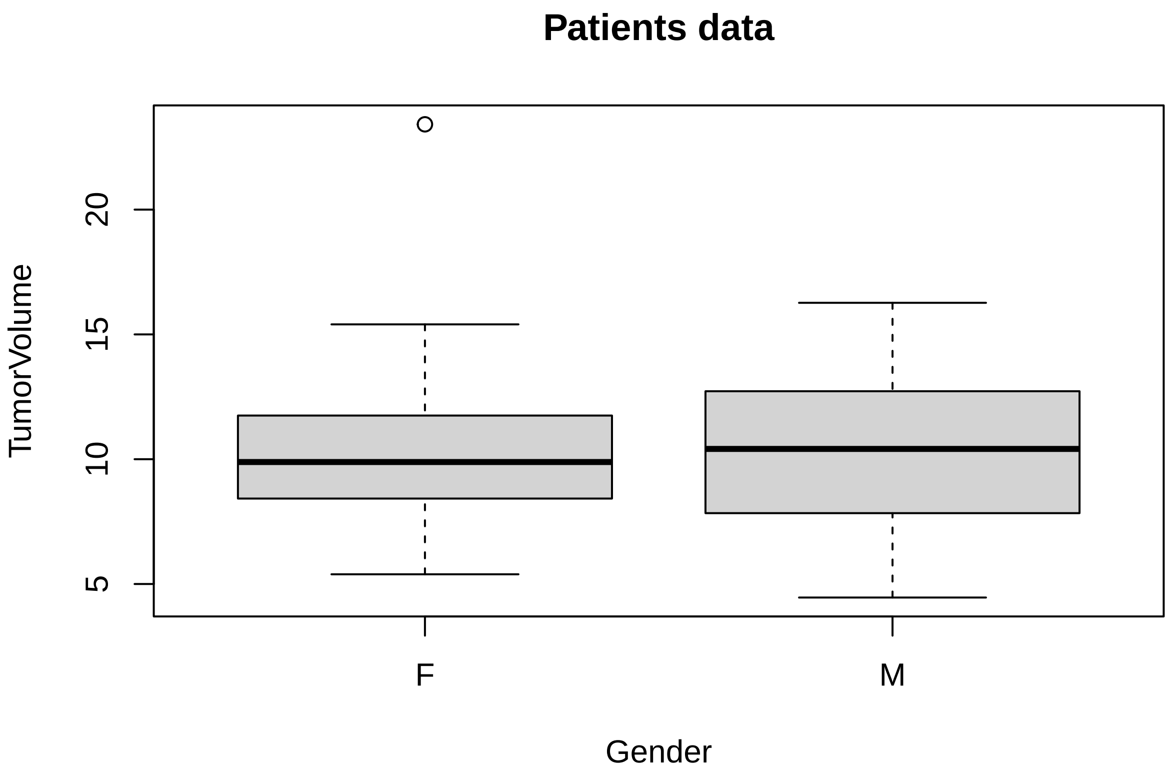
R Biostat. basics

T-test and Wilcoxon rank sum test

- T-test measures the difference in mean between two independent populations [assumes normal distribution]
- Wilcoxon rank sum test measures the difference in the median between two independent populations [non-parametric]

R Biostat. basics

```
boxplot(TumorVolume ~ Gender, patients.df
  , ylab="TumorVolume",xlab="Gender",main="Patients data")
```



R Biostat. basics

Let's use the Wilcoxon rank sum test to see if female and male patients are significantly different in regards to Tumor Volume

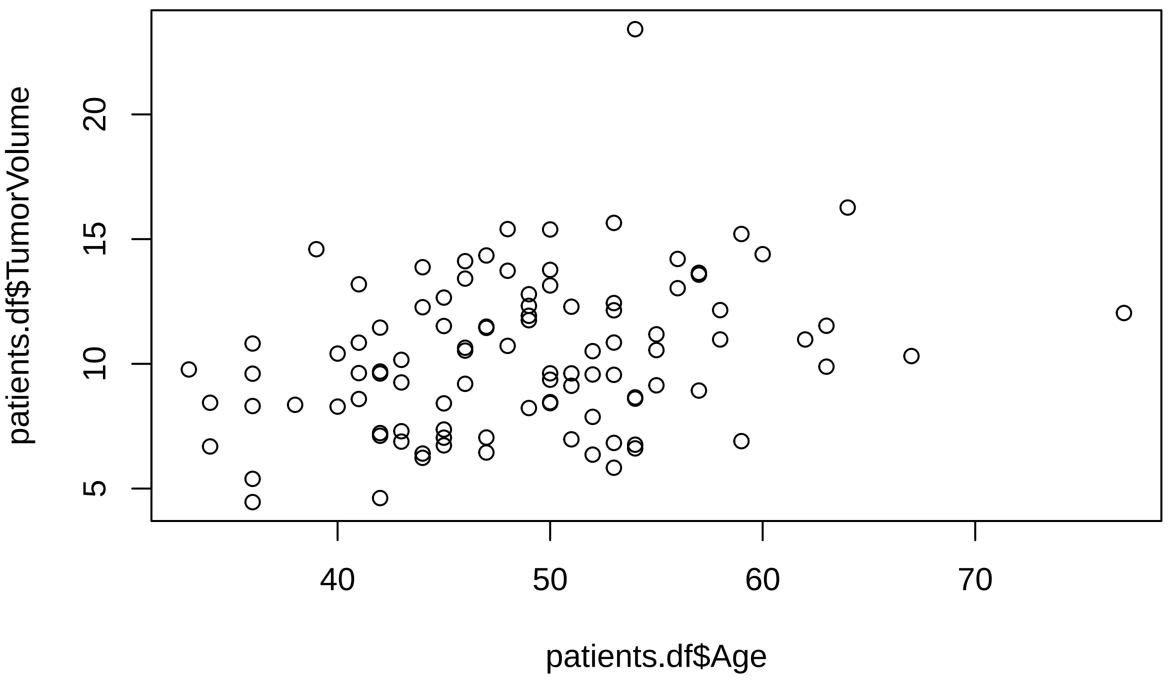
```
  wilcox.test(x = patients.df$TumorVolume[patients.df$Gender=="M"]
              ,y = patients.df$TumorVolume[patients.df$Gender=="F"]
              ,mu = 0,alternative = "two.sided",paired = F,T)
```

```
##
##  Wilcoxon rank sum exact test
##
## data:  patients.df$TumorVolume[patients.df$Gender == "M"] and patients.df$TumorVolume[patients.df$Ge
## W = 1301, p-value = 0.6638
## alternative hypothesis: true location shift is not equal to 0
```

R Biostat. basics

Let's check the correlation between Age and Tumor volume

```
plot(patients.df$Age,patients.df$TumorVolume)
```



R Biostat. basics

- Pearson product moment correlation: The Pearson correlation evaluates the linear relationship between two continuous variables.
- Spearman rank-order correlation: The Spearman correlation evaluates the monotonic relationship between two continuous or ordinal variables.

R Biostat. basics

Let's check the correlation between Age and Tumor volume

```
cor.test(patients.df$Age,patients.df$TumorVolume)
```

```
##
##  Pearson's product-moment correlation
##
## data:  patients.df$Age and patients.df$TumorVolume
## t = 3.1949, df = 98, p-value = 0.001882
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1178229 0.4749021
## sample estimates:
##       cor
## 0.3071332
```

Project

Project

1- Download the following file from: https://drive.google.com/drive/folders/1Jqa5KxKyfYqMMkk1IOwKihOZw-IeedxT

2- Find the if there is a significant difference between TNBC and nonTNBC samples based on these genes

[ESR1, PGR, ERBB2] and show that in a plot. 3- Compute the pairwise correlation between all the samples and show that in a plot.

Hint: Install and use package **corrplot**