Introduction to programming in R: Day 2

Kanhu Charan Moharana

November 26, 2019

Learning objectives

- Subsetting R objects
- Operators used in subsetting
- Functions
  - Built-in and Utility functions
  - Statistical functions

  -The 'apply' family functions
  - apply()
  - sapply()
  - lapply()
- Descriptive statistics using R
- Statistical Plotting using base R

Operators used in Subsetting

- Single square bracket ([ ) :
  - Useful to select one or more elements
  - returns object of same class.
- Double square bracket ([[ ) :
  - Extract elements of a list or a data-frame
  - Returns object of different class
- Dollar operator ($) :
  - Extract elements of a list or a data-frame by Name attribute
  - Similar to [[

Single square bracket ([ ])

```
a <- c('a','e','i','o','u')
a[1]
```

```
## [1] "a"
```

```
a[5]
```

```
## [1] "u"
```

```
a[6]
```

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

```
a[1:4]  # 1:4 returns all numbers from 1 to 4
```

```
## [1] "a" "e" "i" "o"
```

Contd...

Single square bracket ([ ])

Logical vector

- we may also use a logical vector to subset
  - only TRUE indices are returned

```
# we may also use a logical vector to subset
a[c(TRUE, FALSE, TRUE)]
```

```
## [1] "a" "i" "o"
```

```
# Create a logical vector and pass
a[a > 'a']
```

```
## [1] "e" "i" "o" "u"
```

Double square bracket ([[): list

```
l <- list( 1:5, 0.5 )
l[[1]]
```

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

```
l[[2]]
```

```
## [1] 0.5
```

```
l[[3]]
```

```
#l[[3]]
```

Dollar operator ($)

Dollar ($) operator

```
k <- list( foo=1:5, bar=0.5 )
k
```

```
## $foo
## [1] 1 2 3 4 5
##
## $bar
## [1] 0.5
```

```
k$foo
```

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

```
k$bar
```

```
## [1] 0.5
```

```
k$baz
```

```
## NULL
```

Dollar operator ($) vs Double square bracket

Why use "[[" over $?

- "[[" are useful with variables

```
score <- list(
  country=c('Brazil','Germany'),
  goals=c(8,3)
  )
x <- 'country'
score[[x]] ## prints  'Brazil','Germany'
```

```
## [1] "Brazil"  "Germany"
```

```
score$x    ## NULL
```

```
## NULL
```

Subset matrices

- MATRIX [<row> , <column>] : one particular elemnt
- MATRIX [<row_range> , <column0_range>] : elements from the range
- MATRIX [, <column>]: all rows and specified columns
- MATRIX [<row> ,] : all columns of specified rows

```
mat <- matrix(1:15, byrow=T, ncol=3)
mat

mat[1,1] # row, column
mat[3,1]

mat[1, ]  # single row
mat[ ,3]    # single column

mat[ ,3, drop=F]
```

Subset matrices

```
mat <- matrix(sample(1:100, size=8), byrow=T, ncol=2)
colnames(mat) <- c('sample_1','sample_2')
rownames(mat) <- c('Gene1','Gene2','Gene3','Gene4')
mat
```

```
##       sample_1 sample_2
## Gene1       14       39
## Gene2       85       24
## Gene3        1       75
## Gene4       32       93
```

- `MATRIX [ <row name vector> , <column name vector> ]` : select rows and columns
- `MATRIX [ <row name vector> , ]` : select all columns of specified rows
- `MATRIX [ , <column name vector> ]` : select all rows of specified columns Subset matrices

Subset dataframes

- similar as matrix subset
- can use $ to select a column

```
df <- data.frame(
   foo=seq(1,10,by=2),
   bar=seq(0.1, 0.5, by=0.1),
   baz=letters[10:14]
)
df
```

```
##   foo bar baz
## 1   1 0.1   j
## 2   3 0.2   k
## 3   5 0.3   l
## 4   7 0.4   m
## 5   9 0.5   n
```

Subset dataframes

Subset using row index

```
df[1, ]
```

```
##   foo bar baz
## 1   1 0.1   j
```

```
df[1:2, ]
```

```
##   foo bar baz
## 1   1 0.1   j
## 2   3 0.2   k
```

Subset using column index

```
## [1] j k l m n
## Levels: j k l m n
##   foo bar
## 1   1 0.1
## 2   3 0.2
```

Subset dataframes

Subset columns using 'names' attributes

```
# using [ like a vector()
df[, 'foo']
```

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

```
# using [[ like a list()
df[['foo']]
```

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

```
# using dollar like a list()
df$foo
```

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

```
df[, c('foo','bar')]
```

```
##   foo bar
## 1   1 0.1
## 2   3 0.2
```

Subset dataframes

Subset rows using 'row.names' attributes

```
row.names(df)
```

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

```
## Assign a unique column as row.names()
row.names(df) <- df$baz
df
```

```
##   foo bar baz
## j   1 0.1   j
## k   3 0.2   k
## l   5 0.3   l
## m   7 0.4   m
## n   9 0.5   n
```

```
df['k',]
```

```
##   foo bar baz
## l   3 0.2   l
```

Removing missing values

- Empty/missing values: NAs
  - create a logical vector or matrix

```
# in vectors
dat <- c(1,14,NA,8, NA)
bad <- is.na(dat)
bad
```

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

```
dat[bad]
```

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

```
# bang (!) is used for negation
dat[!bad]
```

```
## [1]  1 14  8
```

Removing missing values

1. `complete.cases()`
   - Return a logical vector indicating which cases are complete, i.e., have no missing values

```
mat <- matrix(seq(1,12), ncol=3)
mat[1,3] <- NA
mat[2,2] <- NA
```

```
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    5   NA
## [2,]    2   NA   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
good <- complete.cases(mat)
good
```

Removing missing values

2. `na.omit()`

- returns the object with incomplete cases removed.

```
na.omit(mat)
```

```
##      [,1] [,2] [,3]
## [1,]    3    7   11
## [2,]    4    8   12
## attr(,"na.action")
## [1] 2 1
## attr(,"class")
## [1] "omit"
```

Example: Filter countries with data for each year

```
## Import data as a data frame
dat <- read.csv(
  file='Data/Infant_mortality_rate_UN.csv',
  skip = 1,
  check.names = F
  )
## Rows and columns, also try dim(dat)
nrow(dat)
```

```
## [1] 63
```

```
ncol(dat)
```

```
## [1] 53
```

Contd...

Example:

```
## remove NA rows
dim(na.omit(dat))
```

```
## [1] 24 53
```

```
## Use LOGICAL vector returned by  complete.cases(dat) to subs
dim(
  dat[ complete.cases(dat) , ]
)
```

```
## [1] 24 53
```

na.omit() vs complete.cases()

- na.omit and complete.cases functionally look similar
  - na.omit() : returns the object with filtered values
  - complete.cases(): returns a logical sequence.
  - More: https://stackoverflow.com/questions/29472540/when-to-use-na-omit-versus-complete-cases

Vectorized Operations

- perform computational operation on the vectors in a parallel fashion
- do not require looping

```
## Add 1 to each value in the vector
foo <- c(11,18,100,8)
foo + 1
```

```
## [1]  12  19 101   9
```

```
## Mathematical operation each element by element
a <- c(1,2,5,10)
b <- c(15,3,3,1)
a * b
```

```
## [1] 15  6 15 10
```

Vectorized Operations

- The examples shown can only be executed if the vector or matrices are of equal dimentions.

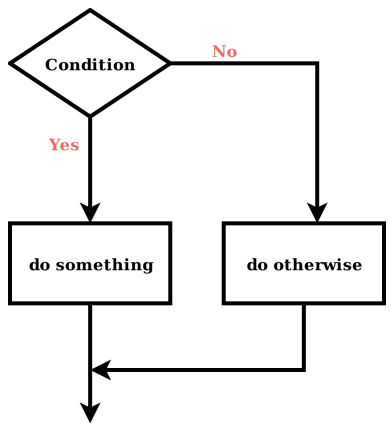
```
m <- matrix(c(10,22,12,18), ncol=2)
n <- matrix(c(22,5,0,2), ncol=2)
m * n
```

```
##      [,1] [,2]
## [1,]  220    0
## [2,]  110   36
```

- This is not matrix multiplication.
  - Mathematical matrix multiplication can be performed using %*%
  - See : https://www.mathsisfun.com/algebra/matrix-multiplying.html

Control structures

- In a program, a control structure determines the order in which statements are executed.
- control execution of the programs
  - if, else
  - `for` loop
  - `while` loop
  - `break`
  - `next`



Logical operations

- We can control execution of some instructions using logic operations by manipulating Boolean values (`TRUE/FALSE, 1/0`).

```
name <- 'Kanhu'

## `==` is used to compare two varaibles and get a Boolean va
name == 'kanhu'
```

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

```
name == 'Kanhu'
```

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

```
name != 'kanhu'
```

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

```
# Numeric value
5 > 10
```

Logical operations

```
## vectorized operation
a <- c(10,5,18,100,NA)
a > 5
```

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

```
## Boolean values can be used for subsetting
a[a>5]
```

```
## [1]  10  18 100  NA
```

Logical operators

Operators	Description
<	Less than
>	Greater than
<=	Less than equal to
>=	Greater than equal to
==	Equal to
!=	Not equal to

Multiple logic

Operator	Description
!	NOT
&	AND
\| | OR |

Logical operators

Truth Tables

- if p and q are two logical comparisons

AND

p	q	p AND q
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

Contd. . .

Logical operators

OR

p	q	p OR q
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

ifelse()

Data	> 10
6	N
9	=IF(C9>10,"Y","N")
13	Y
7	N
9	N
12	Y
5	N
9	N
6	N
4	N
8	N
8	N
5	N
11	
7	

Figure 1: IF() formula in Excel.

- Like excel R, `ifelse()` function can perform condition check.

```
# if any value >10, assign 'Y'
x <- c(5,10,20,22,6,8,19,40,45,2,5)

ifelse(x>10,"Y","N")
```

```
##  [1] "N" "N" "Y" "Y" "N" "N" "Y" "Y" "Y" "N" "N"
```

Example: Blast tabular output

- You have a BLAST output. Import the Hit table in to R and filter those hits with evalue $<$ 10^-5

```
# import BLAST tabular output
BLASTn <- read.csv(
  file='Data/BLAST_hitTable.csv',
  header=FALSE
  )
```

BLASTn

V1	V2	V3	V4	V5	V6	V7	V8	V9	V
TC00002	LR699748.2	95.336	729	29	5	168	894	4354	36
TC00002	LR699748.2	95.062	729	31	5	168	894	12576	118
TC00002	LR699748.2	89.474	152	16	0	26	177	4026	41

```
## [1] 952  12
```

Default Blast tabular output (format 6)

Header	Description
q.seqid	a factor, query (e.g., gene) sequence id
sseqid	a factor, subject (e.g., reference genome) sequence id
pident	numeric vector, percentage of identical matches
length	integer vector, alignment length
mismatch	integer vector, number of mismatches
gapopen	integer vector, number of gap openings
q.start	integer vector, start of alignment in query
q.end	integer vector, end of alignment in query
s.start	integer vector, start of alignment in subject
s.end	integer vector, end of alignment in subject
evalue	numeric vector, expect value
bitscore	numeric vector, bit score

Example: Blast tabular output

```
# set the column headings using 'names()'
names(BLASTn) <- c('qid','sid','pident','length',
                   'mismatch','gapopen', 'q.st','q.en',
                   's.st','s.en', 'evalue','bitscore')
```

Example: Blast tabular output

```
## Filter rows where evalue is less than 1e-5
BLASTn_2 <- BLASTn[BLASTn$evalue < 1e-5, ]

## Evalue Filtered data
dim(BLASTn_2)
```

```
## [1] 589  12
```

```
## Filter rows where evalue is less than 1e-5 AND identity per
BLASTn_3 <- BLASTn[BLASTn$evalue < 1e-5 & BLASTn$pident >90, ]

## Evalue - %identy Filtered data
dim(BLASTn_3)
```

```
## [1] 247  12
```

if..else

- Similar to `ifelse()`, but more powerful
- can perform more than one instructions

```
if(name == 'Kanhu'){
  print('Hello Kanhu')
  print('Multiple statements ')
}else{
  print('Hello stranger')
}
```

```
## [1] "Hello Kanhu"
## [1] "Multiple statements "
```

for() loop function

- "Looping", "cycling", "iterating" or just replicating set of instructions

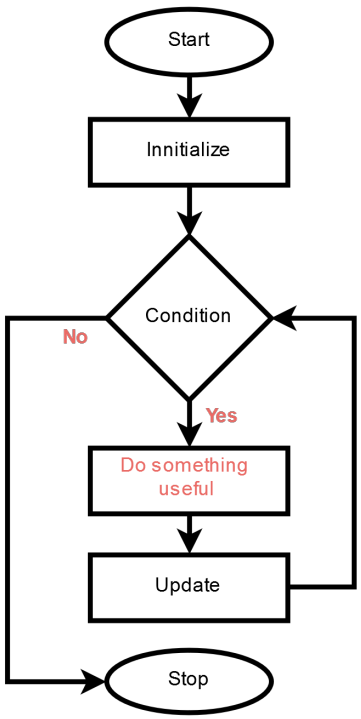


Figure 2: For loop

for() loop function

Syntax

```
for (x in VECTOR)
{
instruction1 to execute using x
instruction2 to execute
instruction-N to execute
}
```

- x will be each value from VECTOR.

for() loop function

```
for(i in 1:4){
  print(i)
}
```

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

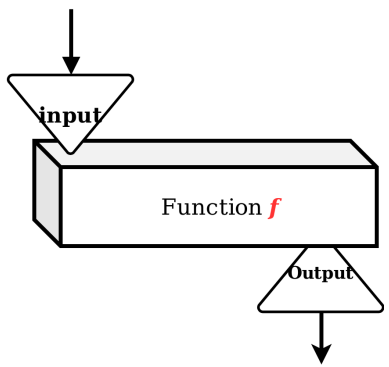
Using for loop to append data to an existing R object

```
credits <- c(0.8,0.5,0.4,0.33,0.9)
# Create a empty numeric vector()
percentage_credit = vector(mode='numeric')
## Sum total of credits
total_credit <- sum(credits)

for(i in 1:length(credits) ){
  ## i will iterate over 1.. length of 'credits' vector
  i_th_perc <- 100 * credits[i]/total_credit
  ## update each index of percentage_credit
  percentage_credit[i] <- i_th_perc
}
percentage_credit
```

```
## [1] 27.30375 17.06485 13.65188 11.26280 30.71672
```

Functions

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
- they accept inputs to process and return output.



https://www.tutorialspoint.com/computer_programming/computer_programming_functions.htm

Built-in functions

- Like Excel, in R there exists several built-in functions.

V1	V2	V3
c()	mean()	read.table()
paste()	write.csv()	print()
read.csv()	seq()	max()
rep()	min()	sd()

Useful Built-in functions

1. seq()

```
# seq(x,y): create a sequence of numbers from x to y
seq(1,5)  # see details using ?seq()
```

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

```
seq(1,10, by=2)
```

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

```
seq(5,-5)
```

```
##  [1]  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

Useful Built-in functions

2. rep()

```
# rep(x, n): replicates the values in x, n times.
rep('foo', 5)  # see details using ?seq()
```

```
## [1] "foo" "foo" "foo" "foo" "foo"
```

```
rep(c('foo','bar'), each =2 )
```

```
## [1] "foo" "foo" "bar" "bar"
```

```
rep(c('foo','bar'), each =2, len=10 )
```

```
##  [1] "foo" "foo" "bar" "bar" "foo" "foo" "bar" "bar" "foo"
```

Useful functions

3. paste()

```
# paste(m,n): concatenate m and n
genes<- c('DMPK', 'ATN1', 'EGFR','FMR1', 'HTT')
paste('At_',genes)
```

```
## [1] "At_ DMPK" "At_ ATN1" "At_ EGFR" "At_ FMR1" "At_ HTT"
```

```
paste('At',genes, sep='_')
```

```
## [1] "At_DMPK" "At_ATN1" "At_EGFR" "At_FMR1" "At_HTT"
```

```
# Convert into one string
paste('At',genes, sep='_', collapse=",")
```

```
## [1] "At_DMPK,At_ATN1,At_EGFR,At_FMR1,At_HTT"
```

Useful Built-in functions

4. strsplit()

```
a <- 'My name is kanhu'
strsplit(a, split=' ')
```

```
## [[1]]
## [1] "My"    "name"  "is"    "kanhu"
```

```
a <- c(
  'My name is kanhu',
  'R is wonderful'
  )
strsplit(a, split=' ')
```

```
## [[1]]
## [1] "My"    "name"  "is"    "kanhu"
##
## [[2]]
## [1] "R"         "is"        "wonderful"
```

Useful functions

5. `summary()`

```
x <- c(1,10,100,0.6,55)

#min(x)
#max(x)
#sum(x)

#mean(x)
#sd(x)
summary(x)
```

```
##     Min. 1st Qu.   Median     Mean 3rd Qu.     Max.
##     0.60     1.00    10.00    33.32    55.00   100.00
```

Useful Built-in functions

summary of numerical tables

```
head(iris, n=3)
```

```
##   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
```

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

```
## [1] 3.057333
```

```
# Over all summary
summary(iris$Sepal.Width)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.000   2.800   3.000   3.057   3.300   4.400
```

The apply family functions

- for() is powerful, but needs a lot of typing.
- just hidden loops

A dataframe

```
df <- data.frame(
  A=sample(1:20,5),
  B=sample(11:200,5),
  C=sample(101:300,5)
  )
```

A	B	C
3	93	192
19	170	136
5	145	168
15	123	145
10	39	224

Calculate sum per row: the `for()` loop way

```
# create empty column 'Sum'
df$Sum = NA

for(i in 1:nrow(df) ){
  # sum values from 1-3 column and assign to sum
  df[i,'Sum'] <- sum(df[i,1:3])
}
df
```

```
##    A   B   C Sum
## 1  3  93 192 288
## 2 19 170 136 325
## 3  5 145 168 318
## 4 15 123 145 283
## 5 10  39 224 273
```

Useful functions

6. `apply()` function

- `apply()` is used to execute a function (can be anonymous one) over the margins of an array.

```
apply(X, MARGIN, FUN, ...)
```

- X: Matrix
- MARGIN:
- 1: Row wise
- 2: Column wise
- FUN: function (can be anonymous)
- ...: parameters to FUN
- returns a vector

Calculate sum per row: apply() way

```
# Create a copy of df
df2 <- df[1:3]

## Use 1 as second argument to iterate over rows
df2$Sum <- apply(df2, 1, sum )
df2
```

```
##     A   B   C Sum
## 1   3  93 192 288
## 2 19 170 136 325
## 3   5 145 168 318
## 4 15 123 145 283
## 5 10  39 224 273
```

Calculate sum per COLUMN: `apply()` way

```
## Use 2 as second argument to iterate over columns
apply(df2, 2, sum )
```

```
##     A     B     C   Sum
##    52   570   865 1487
```

More examples of apply()

Calculate average

```
# Create a copy of df (excluding 'Sum' column)
df2 <- df[1:3]
## Use 1 as second argument to iterate over rows
df2$Mean <- apply(df2, 1, mean )
df2
```

```
##    A   B   C      Mean
## 1  3  93 192  96.00000
## 2 19 170 136 108.33333
## 3  5 145 168 106.00000
## 4 15 123 145  94.33333
## 5 10  39 224  91.00000
```

Contd...

More examples of apply()

Calculate standard deviation

```
df2 <- df[1:3]
## Use 1 as second argument to iterate over rows
df2$SD <- apply(df2, 1, sd )
df2
```

```
##    A   B   C        SD
## 1  3  93 192  94.53571
## 2 19 170 136  79.21069
## 3  5 145 168  88.22131
## 4 15 123 145  69.57969
## 5 10  39 224 116.09048
```

Contd...

More examples of apply()

Calculate 25th and 75th quantiles

```
# Create a copy of df
df2 <- df[1:3]
## Use ?quantile to see details about quantile()
## - prob=c(0.25,0.75) is a parameter for quantile()
apply(df2, 1, quantile, prob=c(0.25,0.75) )
```

```
##      [,1]  [,2]  [,3] [,4]  [,5]
## 25%  48.0  77.5  75.0   69  24.5
## 75% 142.5 153.0 156.5  134 131.5
```

Useful functions

7. `lapply()`: loop over lists

- `lapply()` loops over a list and evaluates a function.
- always returns a list

```
x <- list(
  a = 1:20,
  b = seq(0,1, length=10),
  c = seq(1,100, length=14)
)
x
```

```
## $a
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##
## $b
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.55
##  [8] 0.7777778 0.8888889 1.0000000
##
```

lapply(): loop over lists

```
## calculate mean of each objects in the list x
lapply(x, mean)
```

```
## $a
## [1] 10.5
##
## $b
## [1] 0.5
##
## $c
## [1] 50.5
```

```
## calculate mean of each objects in the list x
lapply(x, summary)
```

```
## $a
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

Useful functions

8. sapply(): loop over vectors

- `sapply()` is similar to *lapply()*, but it returns a vector

```
x <- c(15,105,150,199)
## calculate mean of each objects in the list x
sapply(x, mean)
```

```
## [1]  15 105 150 199
```

```
labs <- c('LBR','CCTH', 'LQFPP')
## add /UENF to each lab
sapply(labs, paste, 'UENF', sep='/')
```

```
##           LBR          CCTH         LQFPP
##   "LBR/UENF"  "CCTH/UENF" "LQFPP/UENF"
```

Useful functions

9. `sample()`

- Select N random samples from a given vector
  - https://web.ma.utexas.edu/users/parker/sampling/repl.htm
  - https://en.wikipedia.org/wiki/Nonprobability_sampling

```
sample(
  x    =  input vector of one elements ,
  size=  the number of items to choose,
  replace = FALSE sampling be with replacement? ,
  prob = vector of probability weights for input elements
  )
```

```
x <- c(1,15,18,3,-9,11)
sample(x, size=3)
```

```
## [1]  3 15 11
```

```
sample(x, size=3, replace=T)
```

Useful functions

9. sample()

```
x <- c(1,15,18,3,-9,11)
# assign probability  to each element of x
p <- c(0.1,0.9,0.1,0.3,0.5,0.1)

sample(x, size=3, prob=p)
```

```
## [1] 15  1 -9
```

```
sample(x, size=3, prob=p, replace=T)
```

```
## [1] 15 15 15
```

```
sample(x, size=4, prob=p, replace=T)
```

```
## [1] 18 -9 -9 15
```

Useful functions

10. `rnorm()`

- generates random numbers following normal distribution with user defines `mean` and `standard deviation`

```
rnorm(
  n = number of observations,
  mean = desired mean value (default 0),
  sd = desired standard deviation value (default 1),
  )
```

Contd...

Useful functions

10. rnorm()

Example

```
random_number <- rnorm(n=20,
                       mean=3,
                       sd=8)


# length
length(random_number)
```

```
## [1] 20
```

```
# Mean
mean(random_number)
```

```
## [1] 3.511735
```

```
# SD
sd(random_number)
```

Descriptive Statistics and Data Visualization

- data visualization is an important part of statistics.
- enables to spot trends and relationships
- make sense of huge amounts of data so that you can take decision
- Central tendency
  - Mean, median
- Data Spread
  - Standard deviation
  - variance
  - inter quartile range
  - median absolute deviation

Frequency data

- Categorical data are often summarized as frequency tables

```
toss <- c('H','T','T','H','T','H','T','T','H','T','H','T','H',
table(toss)
```

```
## toss
## H T
## 8 9
```

Frequency data from data frames

Pokemon data

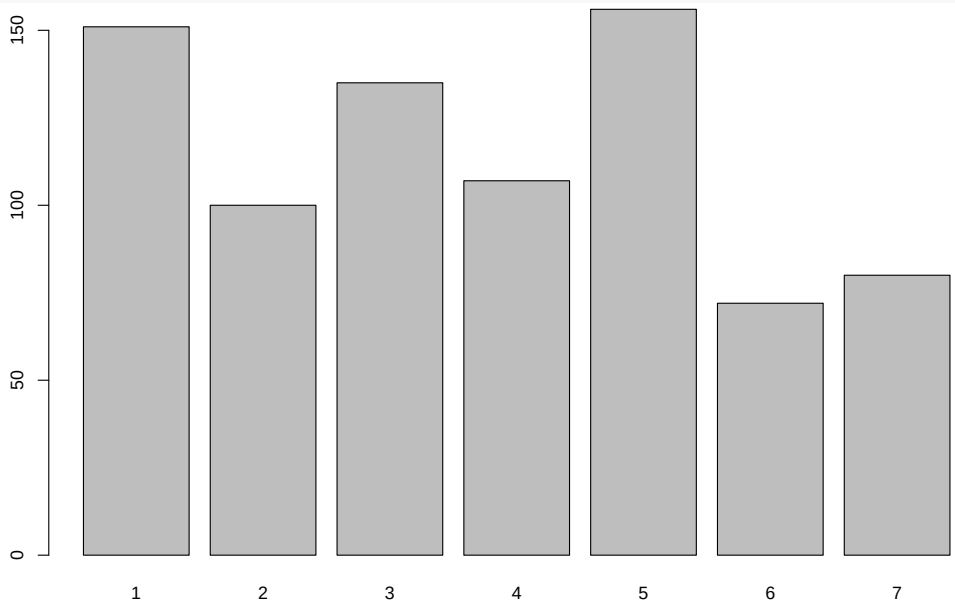
```
# Read the *.csv file // verify your working directory
pokemon_data <- read.csv(
  file='Data/pokemon_data.csv',
  sep=',',
  header=TRUE
)

# Number of Creatures per generatin
table(pokemon_data$generation)
```

```
##
##   1   2   3   4   5   6   7
## 151 100 135 107 156  72  80
```

Visualize Frequencydata/categorical data

1. Bar plot

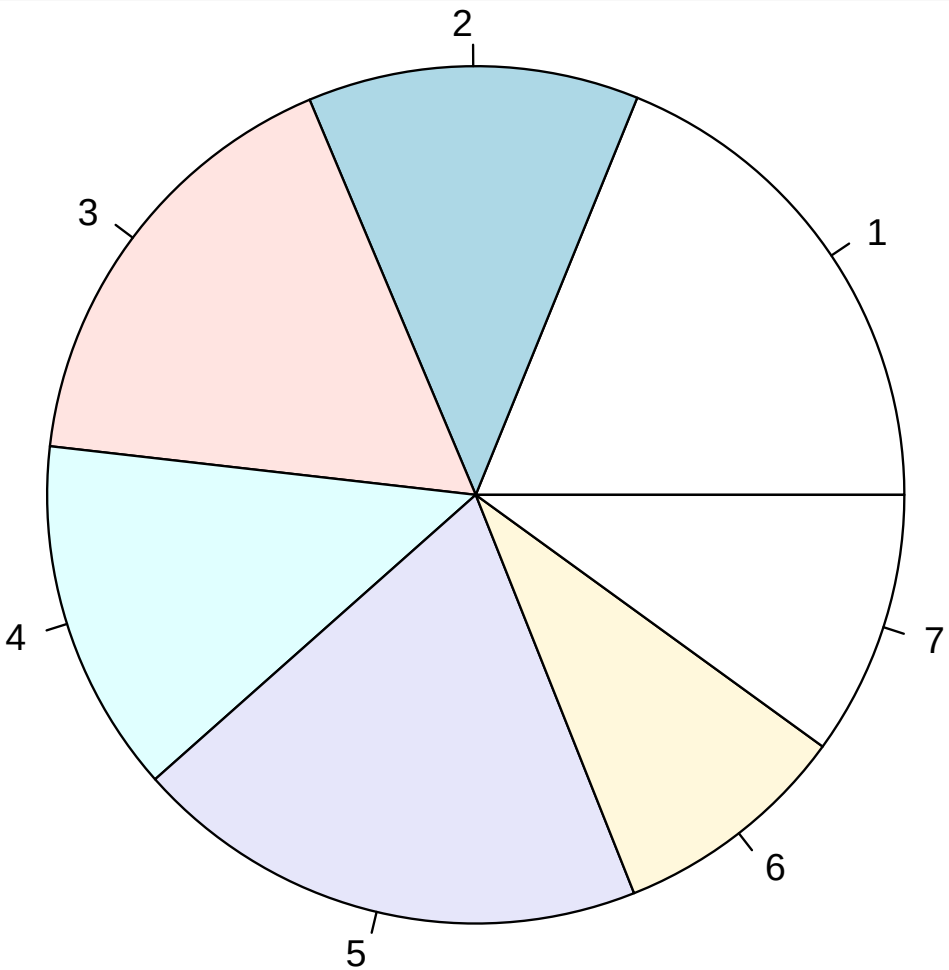
```
barplot(table(pokemon_data$generation))
```



Visualize Frequencydata/categorical data

2. Pie chart

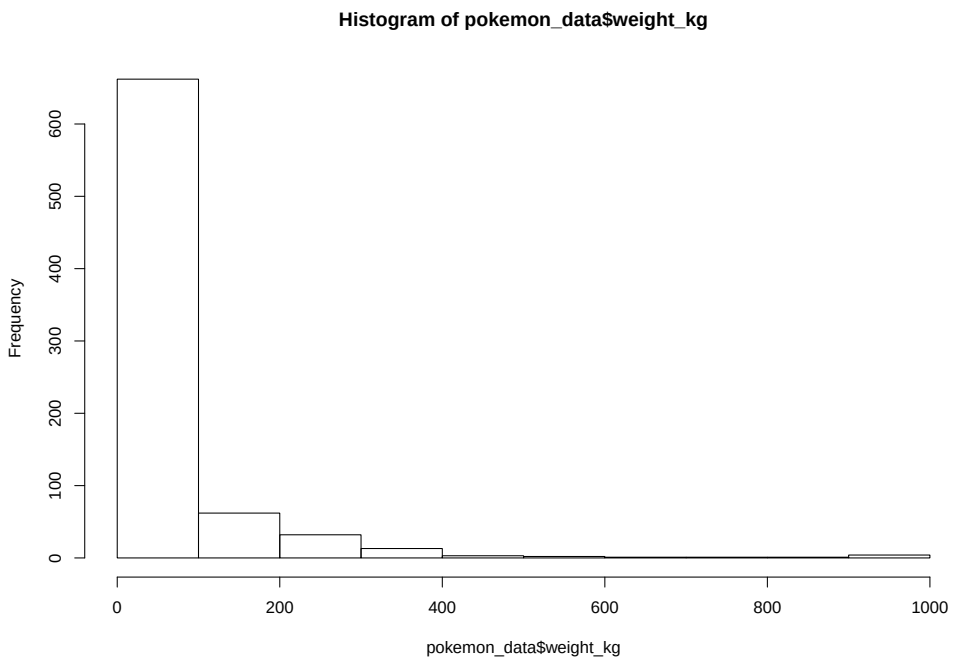
```
pie(table(pokemon_data$generation))
```



Visualize a distribution: histogram

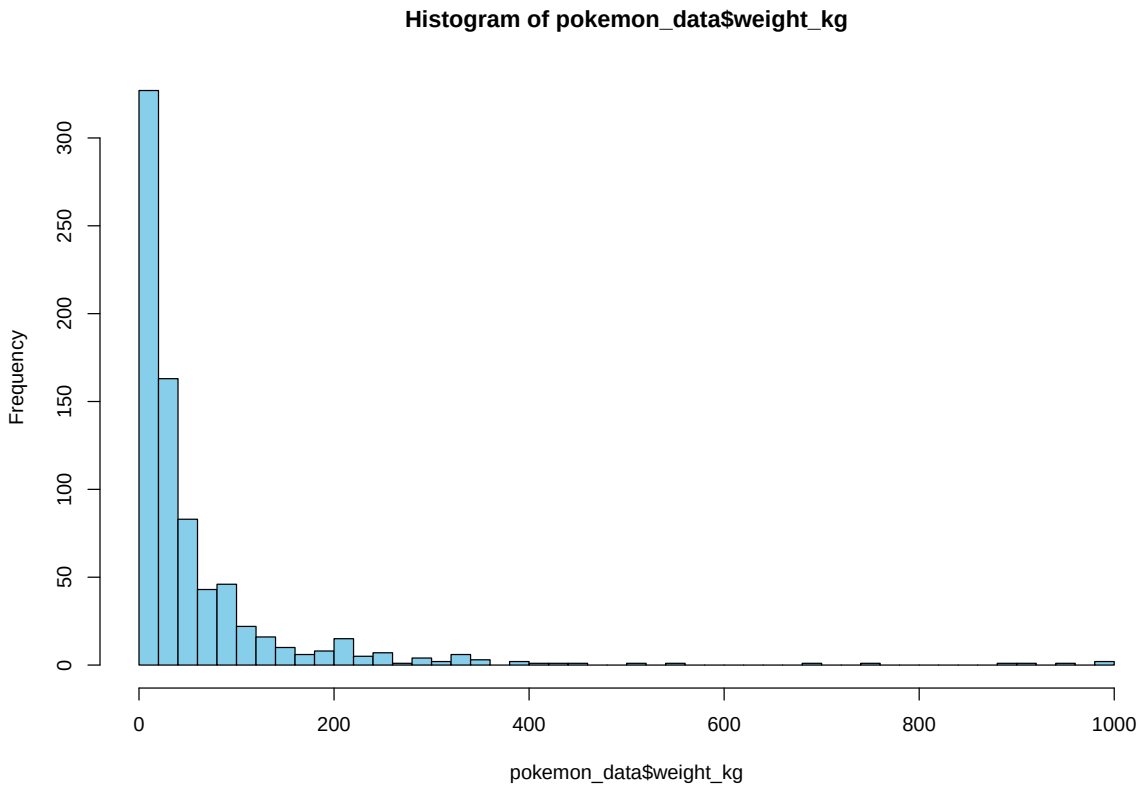
- In a histogram, each bar groups input numbers into ranges or bins or windows.
- Taller bars show that more data falls in that range.

```
hist(pokemon_data$weight_kg)
```



Visualize a distribution: histogram

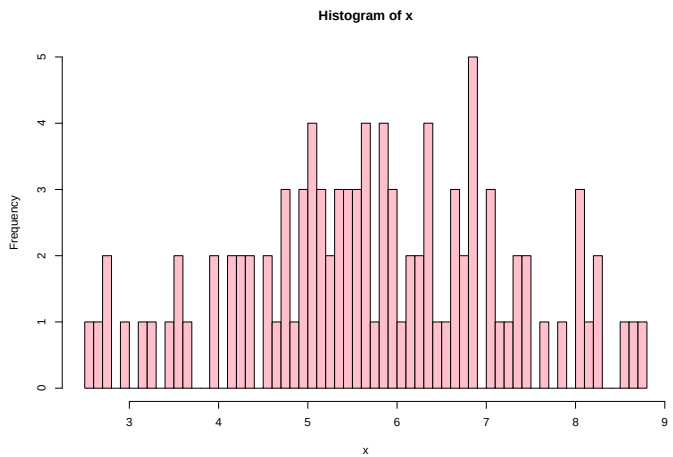
```
# breaks = 50  ### increase number of windows to 50
# col='skyblue' ### fill bars with color, try magenta, teal,
hist(pokemon_data$weight_kg, breaks = 50, col='skyblue')
```



Visualize a distribution: probability distribution

- A probability distribution of a discrete variable, Y, assigns a probability to each possible outcome.
  - https://doi.org/10.1016/B978-0-12-384864-2.00003-2

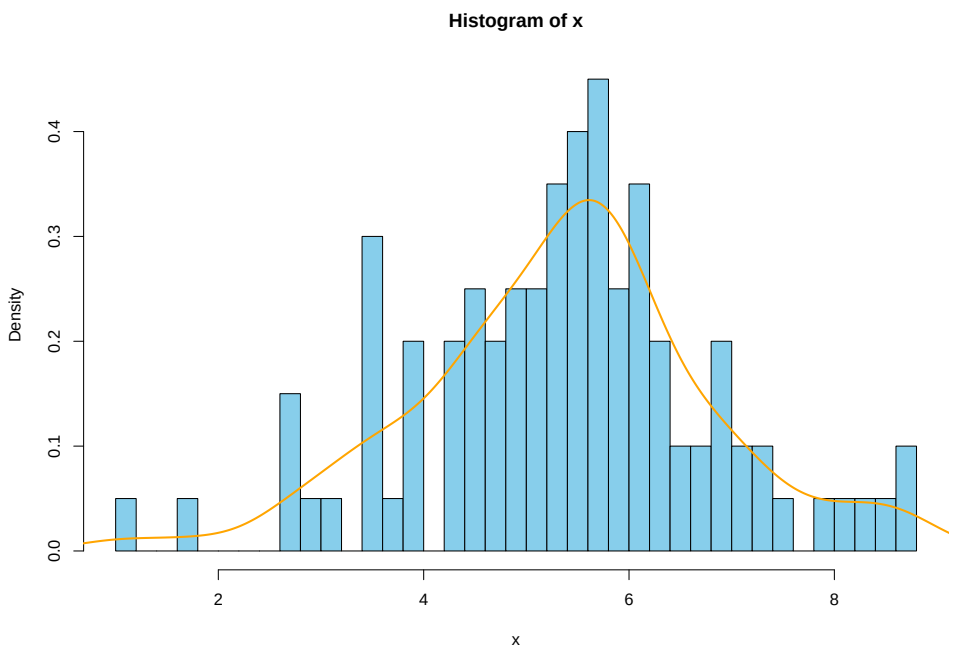
```
## Create a random distribution of 100 numbers
x <- rnorm(n=100, mean=5.5, sd=1.5)

## Run to view Frequency distribution plot
hist(x, breaks = 50,col='pink')
```



https://www.probabilisticworld.com/

Visualize a distribution: probability distribution

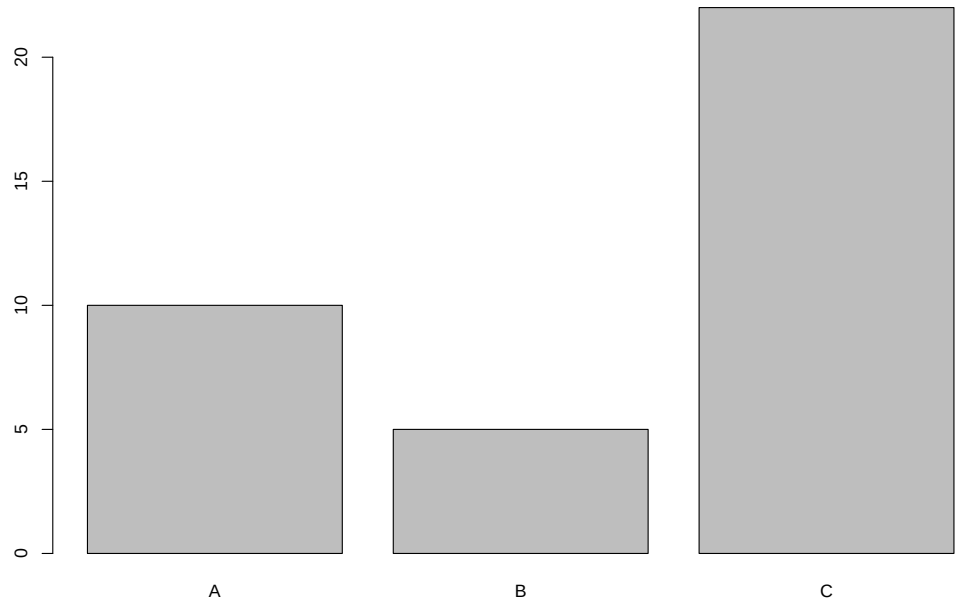
```
## Create a random distribution of 100 numbers
x <- rnorm(n=100, mean=5.5, sd=1.5)
## Density plot
hist(x, breaks = 50, freq = F, col='skyblue')
lines( density(x), col='orange', lwd=2)
```



Barplots

- A barplot (or barchart) shows the relationship between a numerical and a categorical variable.
- Each entity of the categoric variable is represented as a bar. The size of the bar represents its numeric value.

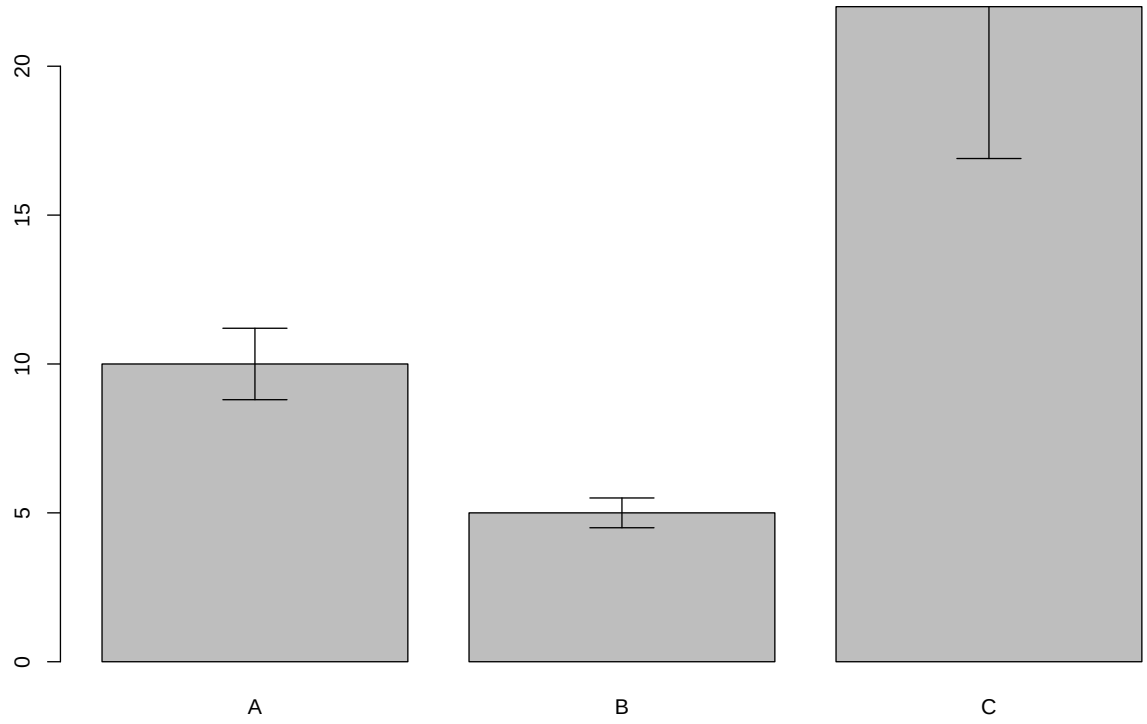
```
df <- data.frame(
  sample=c('A','B','C'),
  mean=c(10,5,22),
  sd=c(1.2,0.5,5.1)
)
barplot(height =df$mean, names.arg=df$sample )
```


0
5
10
15
20
A
B
C

Barplots with Error bars

- Error bars give a general idea on precision of observed value, or conversely, how far from the reported value the true (error free) value might be.

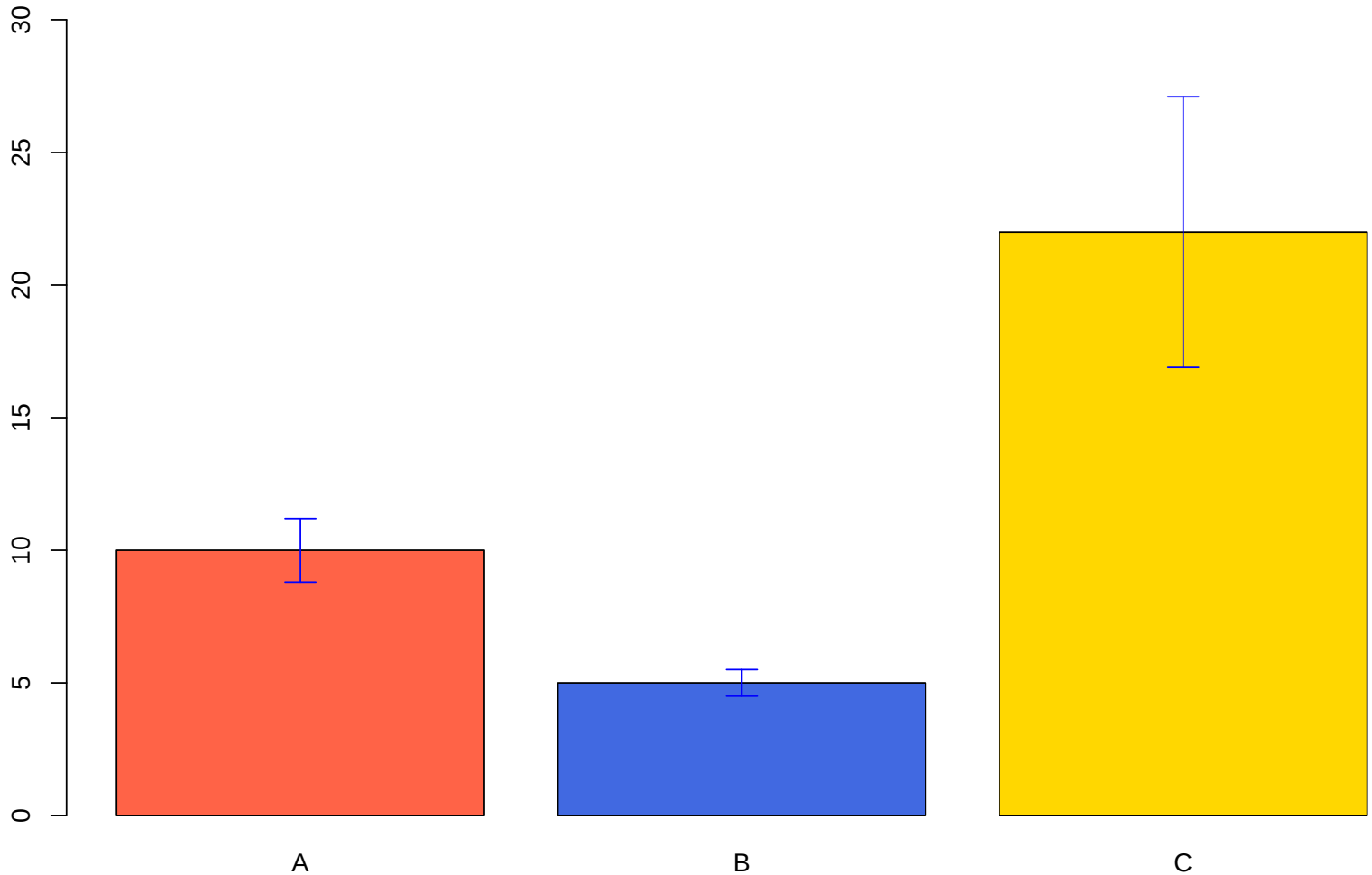
```
bar <- barplot(height =df$mean, names.arg=df$sample)

## arrows() can help to put error bars on  barplots
arrows(x0=bar,
       y0=df$mean-df$sd,
       x1=bar,
       y1=df$mean+df$sd,
       angle=90,
       code=3
       )
```



Contd. . .

Barplots with Error bars

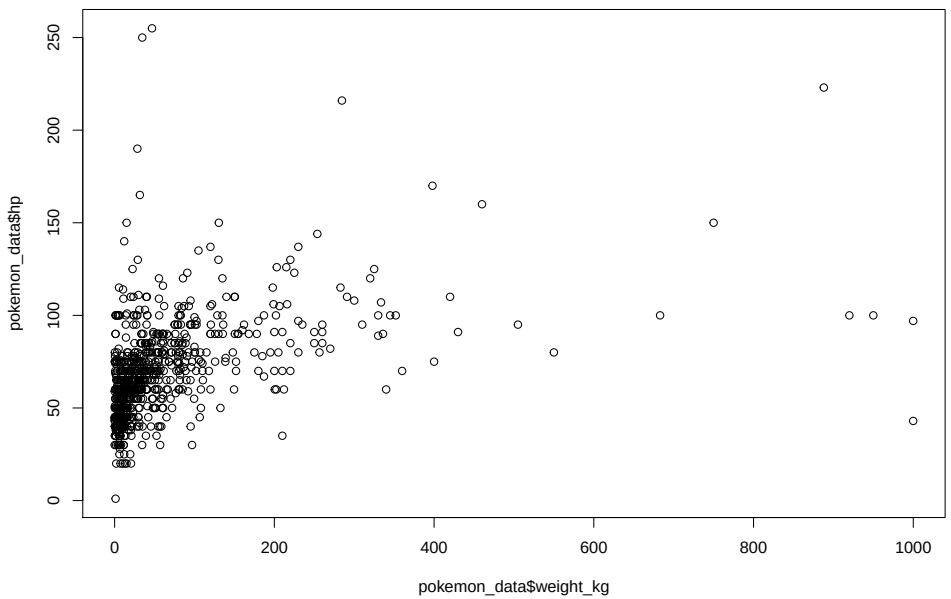
```
bar <- barplot(height =df$mean, names.arg=df$sample ,
               ylim=c(0, 30),
               col=c('tomato','royalblue','gold')
               )
arrows(x0=bar,
       y0=df$mean-df$sd,
       x1=bar,
       y1=df$mean+df$sd,
       angle=90,
       code=3,
       length=0.1,
       col='blue'
       )
```


30
25
20
15
10
5
0
A
B
C

Scatter plots

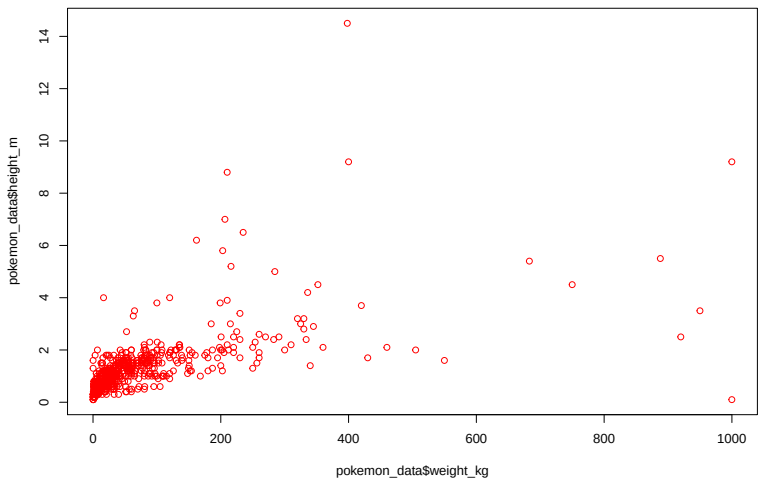
- A scatter plot (aka scatter chart, scatter graph) uses dots to represent values for two different numeric variables.

```
## Compare relationship between weight and HP score.
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp)
```



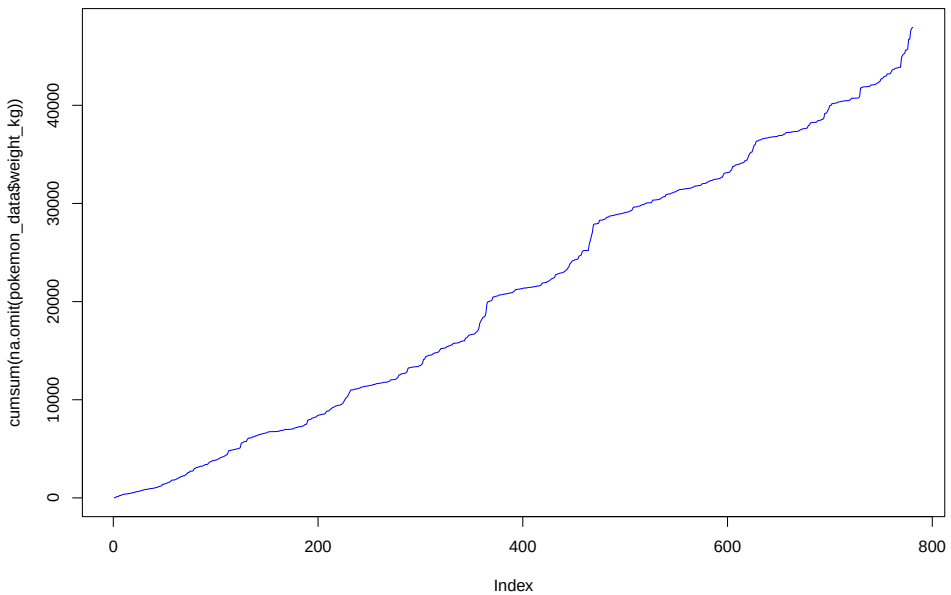
Scatter plot

```
plot(x=pokemon_data$weight_kg,
     y=pokemon_data$height_m,
     col='red'  # Use col to fill color.
    )
```



Line plots

```
plot(cumsum(na.omit(pokemon_data$weight_kg)),
     type='l',    ## default is 'p' to plot points
     col='blue'
     )
```



Line plots

```
#lwd : line width in pixels
plot(cumsum(na.omit(pokemon_data$weight_kg)),
     type='l', col='blue',
     lwd=2   ## line width
     )

#lty : line type: dashed
plot(cumsum(na.omit(pokemon_data$weight_kg)),
     type='l', col='blue',
     lty=3  ## line dash type
     )
```

Boxes and whiskers Plot (Boxplot)

- presents information from a **five-number summary**.
  1. minimum value
  2. lower quartile (Q1)
  3. median value (Q2)
  4. upper quartile (Q3) and
  5. maximum value.

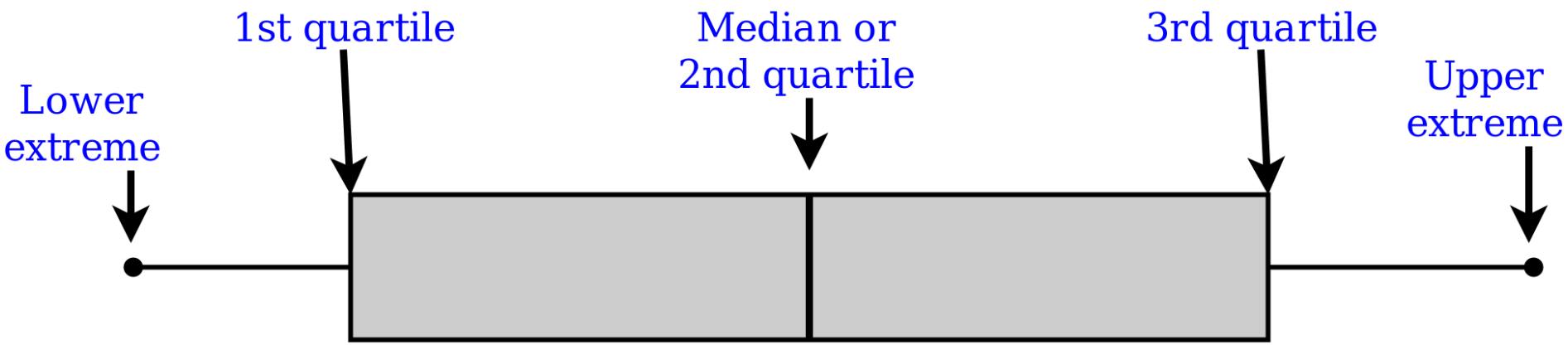


Figure 3: Boxplot

Boxes and whiskers Plot (Boxplot)

`fivenum()` : numerical way

```
#Tukey's five number summary:
#    minimum,
#    lower-hinge,
#    median,
#    upper-hinge,
#    maximum

fivenum(pokemon_data$height_m)
```

```
## [1]  0.1  0.6  1.0  1.5 14.5
```

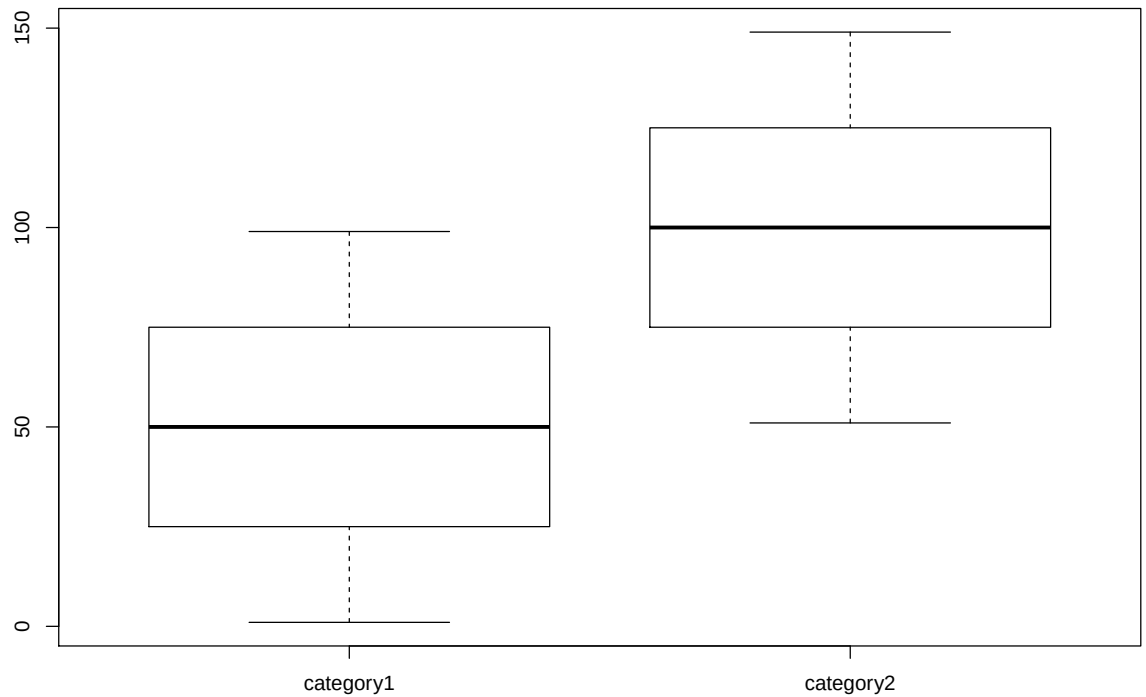
Boxes and whiskers Plot (Boxplot)

- Data from different Categories can be separated by different columns

```
df <- data.frame(
          category1=seq(1,100,by=2),
          category2=seq(51,150,by=2)
    )
# head(df)
summary(df)  ## try fivenum(df$category1)
```

```
##    category1       category2
##  Min.   : 1.0   Min.   : 51.0
##  1st Qu.:25.5   1st Qu.: 75.5
##  Median :50.0   Median :100.0
##  Mean   :50.0   Mean   :100.0
##  3rd Qu.:74.5   3rd Qu.:124.5
##  Max.   :99.0   Max.   :149.0
```

```
boxplot(df)
```



Boxes and whiskers Plot (Boxplot)

```
# logical vector, where type1 column is equal to 'normal'
normal <- pokemon_data$type1=='normal'
grass <- pokemon_data$type1=='grass'   # logical vector
sum(normal)
```

```
## [1] 105
```

```
sum(grass)
```

```
## [1] 78
```

```
df <- list(
  'Normal'=pokemon_data[normal, 'weight_kg'],
  'Grass'=pokemon_data[grass, 'weight_kg'],
  col=c('green','gold')
  )
```

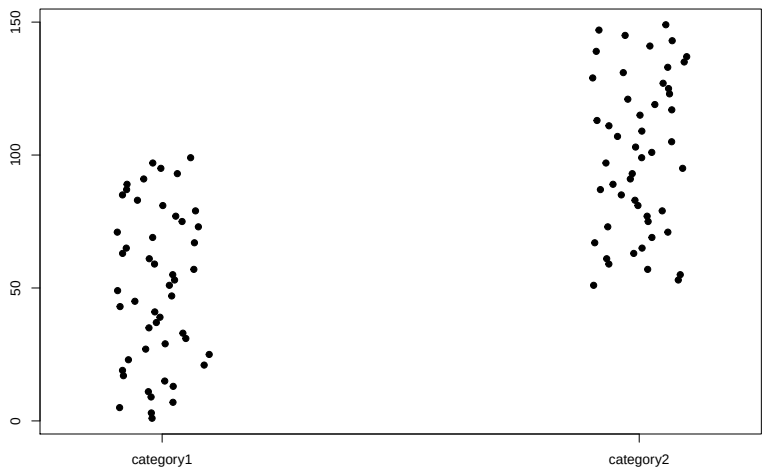
Stripchart

- Stripchart produces one dimensional scatter plots (or dot plots) of the given data. - These plots are a good alternative to boxplots when sample sizes are small

```
##    category1       category2
##  Min.   : 1.0   Min.   : 51.0
##  1st Qu.:25.5   1st Qu.: 75.5
##  Median :50.0   Median :100.0
##  Mean   :50.0   Mean   :100.0
##  3rd Qu.:74.5   3rd Qu.:124.5
##  Max.   :99.0   Max.   :149.0
```

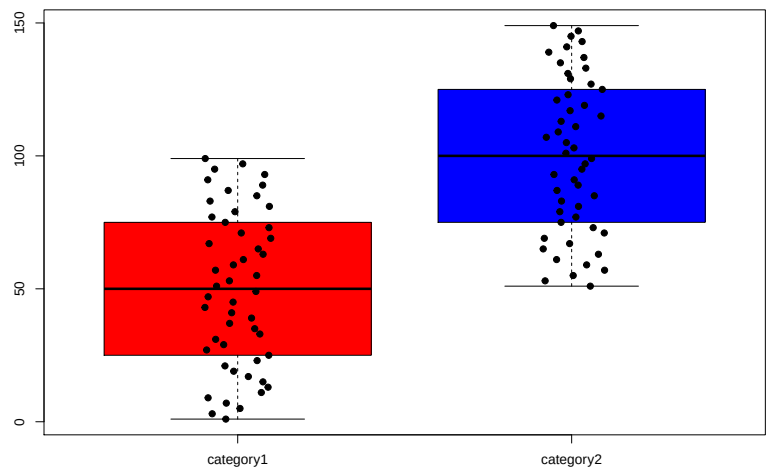
Stripchart

```
stripchart(df, vertical=T, method='jitter', pch=19)
```



Stripchart on boxplot

```
boxplot(df, col=c('red','blue') )
stripchart(df, vertical=T, method='jitter', pch=19, add=T)
```



Various options to improve plots

```
plot(x = 1:10,          # x-coordinates
     y = 1:10,          # y-coordinates

     type = "p",           # Just draw points (no lines)

     main = "My First Plot",  # Plot title

     xlab = "This is the x-axis label",
     ylab = "This is the y-axis label",

     xlim = c(0, 11),         # Min and max values for x-axis
     ylim = c(0, 11),         # Min and max values for y-axis

     col = "blue",    # Color of the points
     pch = 16,    # Type of symbol (16 means Filled circle)
     cex = 1          # Size of the symbols
```

Various options to improve plots

Limit axis range

```
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     xlim=c(0,200),
     ylim=c(0,500) )
```

Axis labels

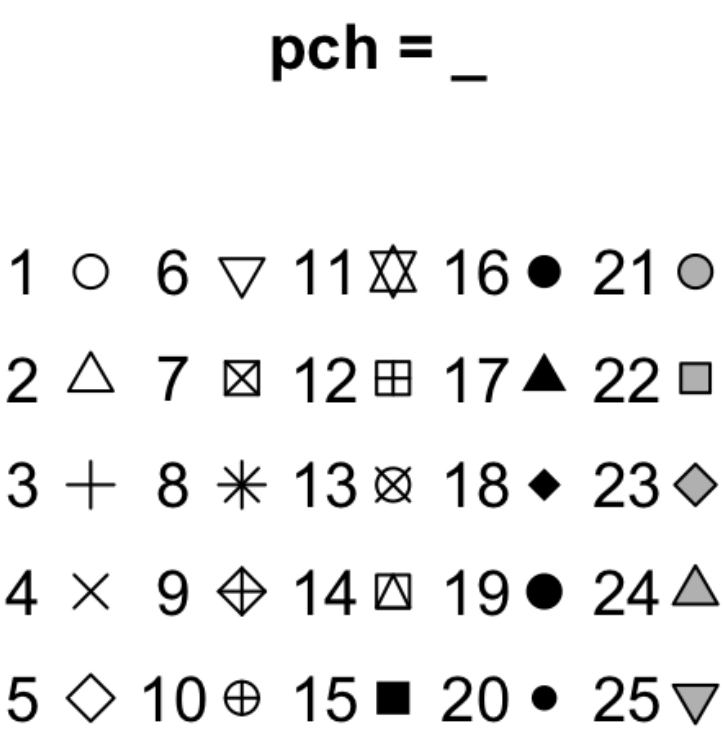
```
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     xlab="Weight (in Kg)",
     ylab="Height (in Mt)" )
```

Plot title

```
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     title='Weight vs Height')
```

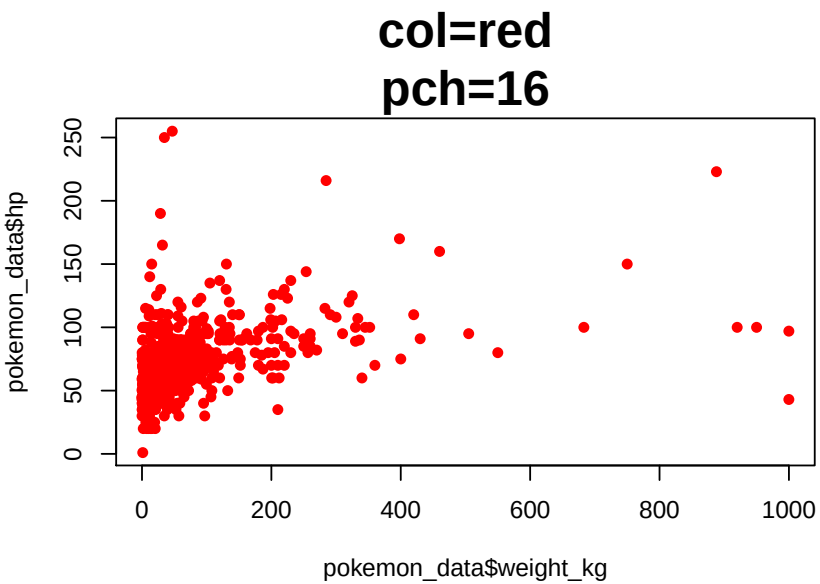
Pointer color and symbols

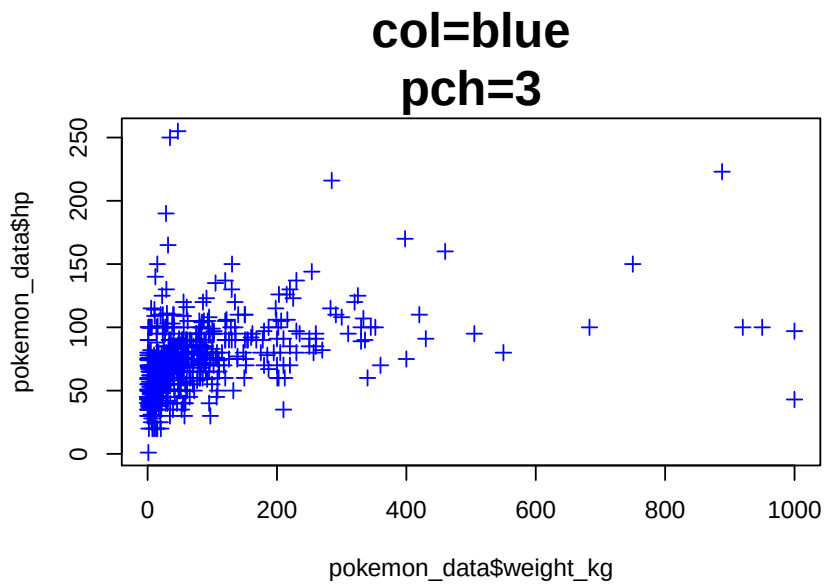
```
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     col='red')
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     col=c('red','blue')  )
```

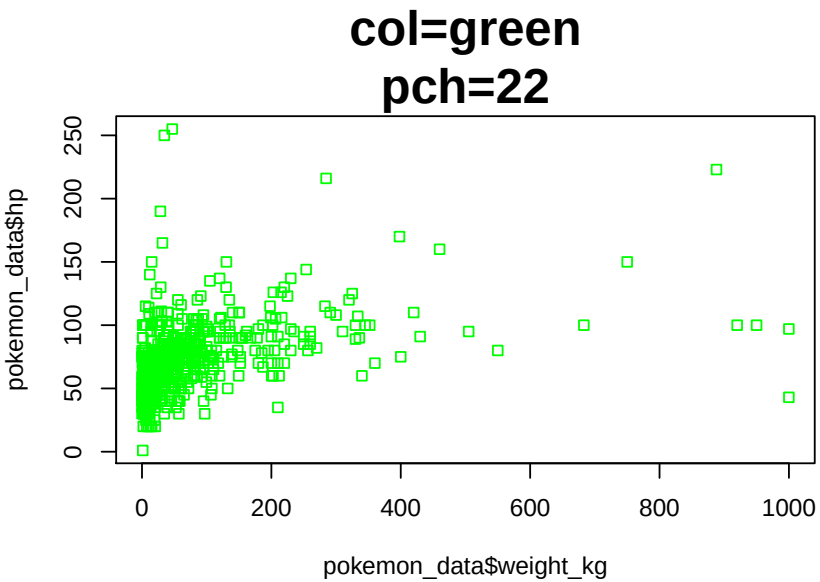


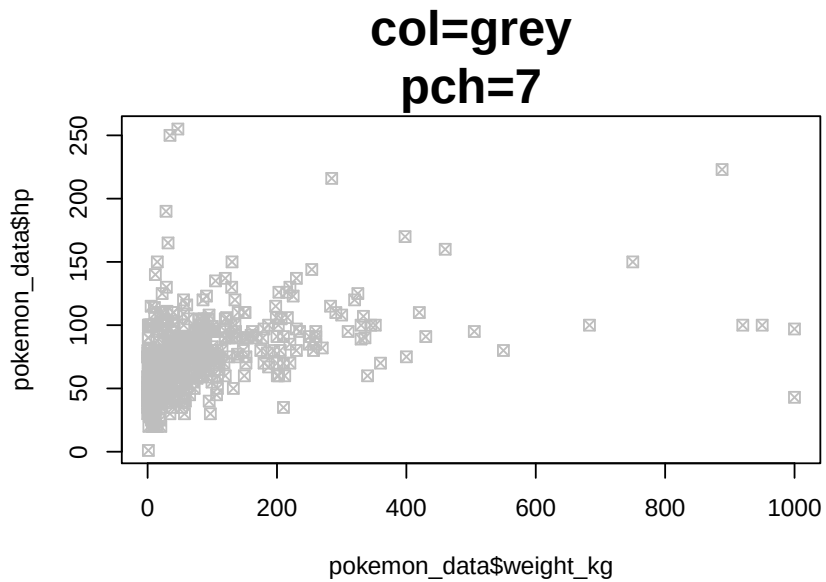
Pointer symbols

```
plot(x=pokemon_data$weight_kg, y=pokemon_data$hp,
     col='red', pch=16)
## change Pch to 3, 22, 7
```


col=red
pch=16
pokemon_data$hp
pokemon_data$weight_kg
0
50
100
150
200
250
0
200
400
600
800
1000


col=blue
pch=3
pokemon_data$hp
pokemon_data$weight_kg
0
50
100
150
200
250
0
200
400
600
800
1000


col=green
pch=22
pokemon_data$hp
pokemon_data$weight_kg
0
50
100
150
200
250
0
200
400
600
800
1000


col=grey
pch=7
pokemon_data$hp
pokemon_data$weight_kg
0
50
100
150
200
250
0
200
400
600
800
1000

Annotation layers

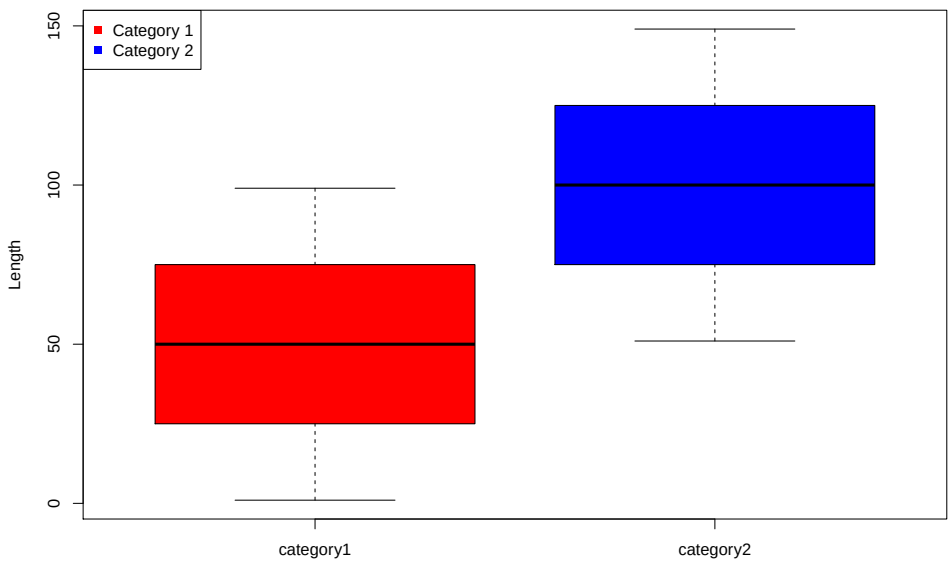
- Some plotting functions can be used for annotation.
  - these are called lower level plot functions.
- they require an existing `plot`

1. `lines(x,y)`
2. `points(x,y)`
3. `text(x,y)`
4. `arrows(x0,y0)`
5. `segments(x0,y0)`
6. `legend()`

Add legend to plots

```
df <- data.frame(
            category1=seq(1,100,by=2),
            category2=seq(51,150,by=2)
    )
# head(df)
```

```r
boxplot(df, col=c('red','blue'), ylab='Length' )
legend('topleft',  ## location on plot
       legend=c('Category 1','Category 2'), ## text
       col=c('red','blue'), ## color
       pch=15
       )
```



Heatmaps

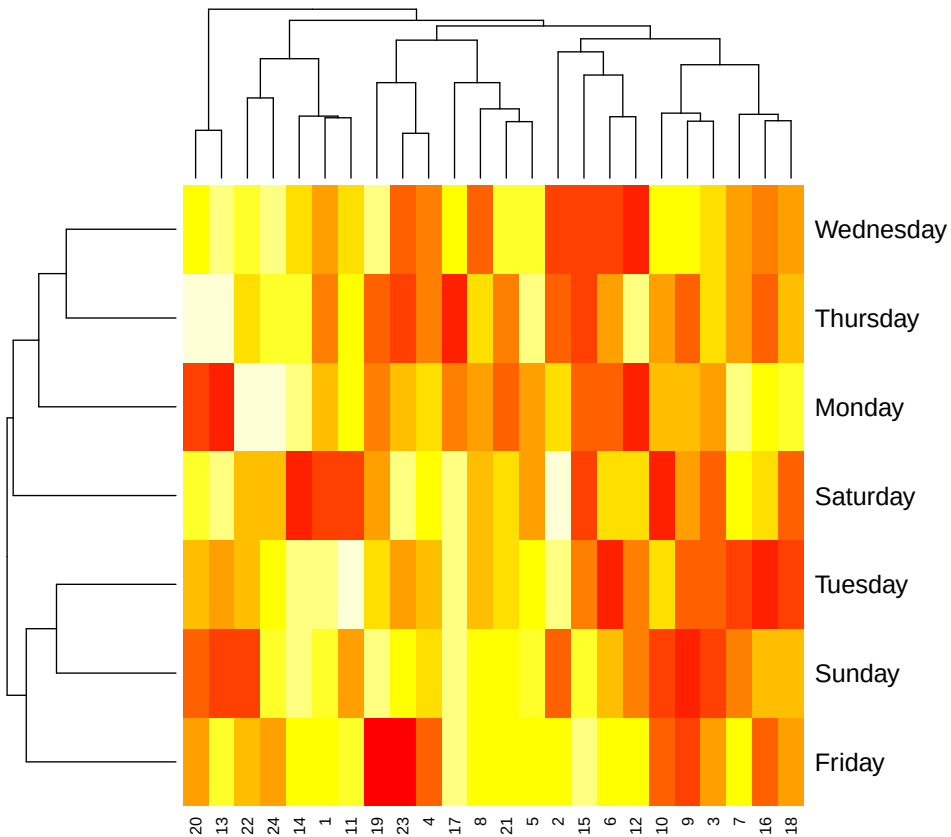
- A heat map (or heatmap) is a graphical representation of data where values are depicted by color.
- Heat maps make it easy to visualize complex data and understand it at a glance

Visitors per Hour per Day

Day	1	2	3	4	5	6	7	8	9	10	11	12	13
Sunday	18	4	2	14	18	10	6	15	0	2	9	6	1
Monday	10	12	9	12	9	4	18	9	11	10	14	0	0
Tuesday	19	18	5	11	15	1	4	11	6	14	20	8	9
Wednesday	8	1	12	5	15	2	7	4	14	13	11	0	18
Thursday	7	5	12	6	16	8	8	12	4	8	13	16	18
Friday	17	17	11	6	17	17	16	16	5	6	18	16	18
Saturday	3	20	5	14	8	13	14	11	9	0	2	13	19

Heatmaps

```
## Import data
# data <- read.csv('Data/visitors.csv', check.names =F)

heatmap(as.matrix(data))
```



Contd. . .

Heatmaps

Meaning of each block

- Each column is a variable.
- Each observation is a row.
- Each square is a value, the closer to yellow the higher.

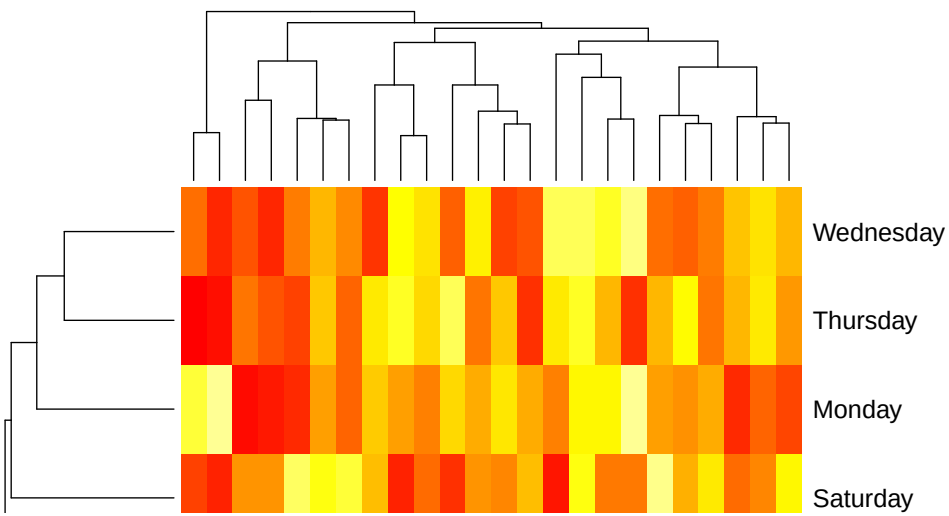
`gplots` library

- use this libray for `heatmap.2()` function

Heatmaps:Color palette

- use the native palettes of R:
  - `terrain.colors()`
  - `rainbow()`
  - `heat.colors()`
  - `topo.colors()`
  - `cm.colors()`

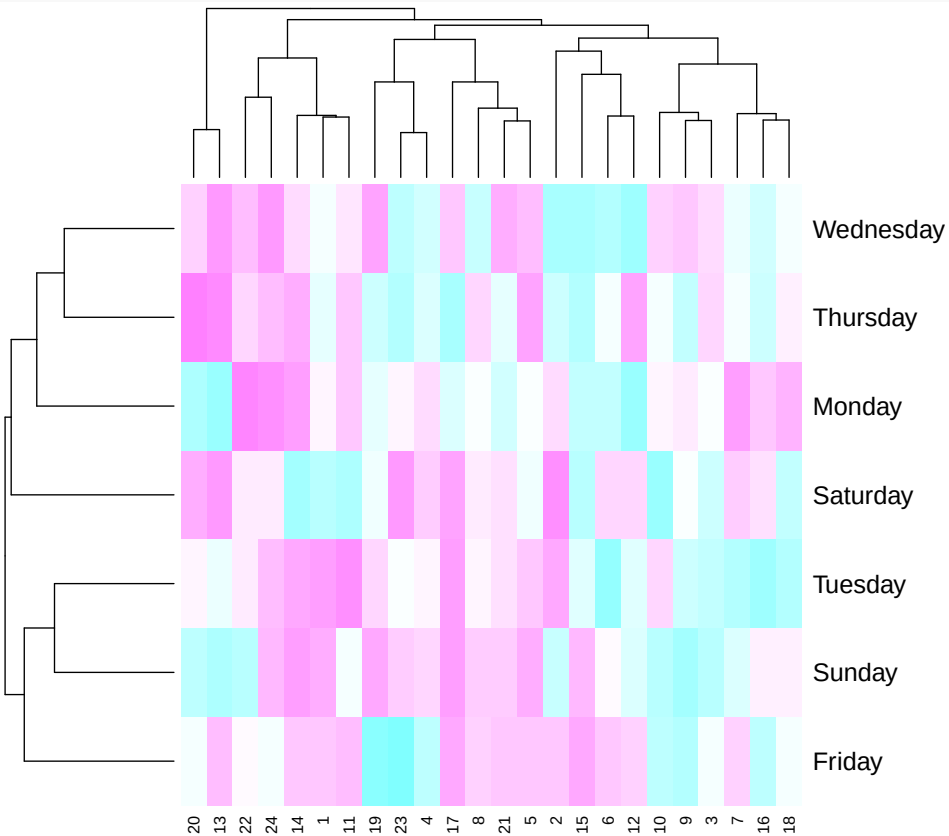
```
## default color shows yellow as highest number, red as lowes
### - rev() is a revese a vector
heatmap(as.matrix(data), col=rev(heat.colors(100)) )
```



Heatmaps:Color palette

col=cm.colors(50)

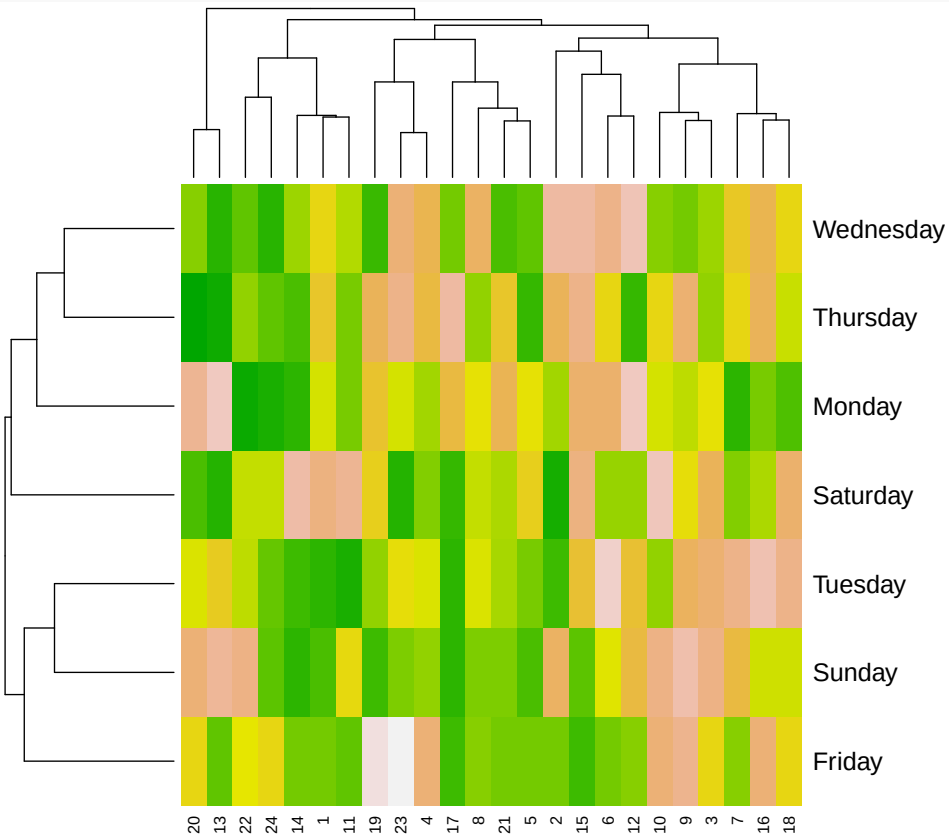
```
heatmap(as.matrix(data), col=cm.colors(50))
```



Heatmaps:Color palette

col=terrain.colors(100)

```
heatmap(as.matrix(data), col=rev(terrain.colors(100)) )
```

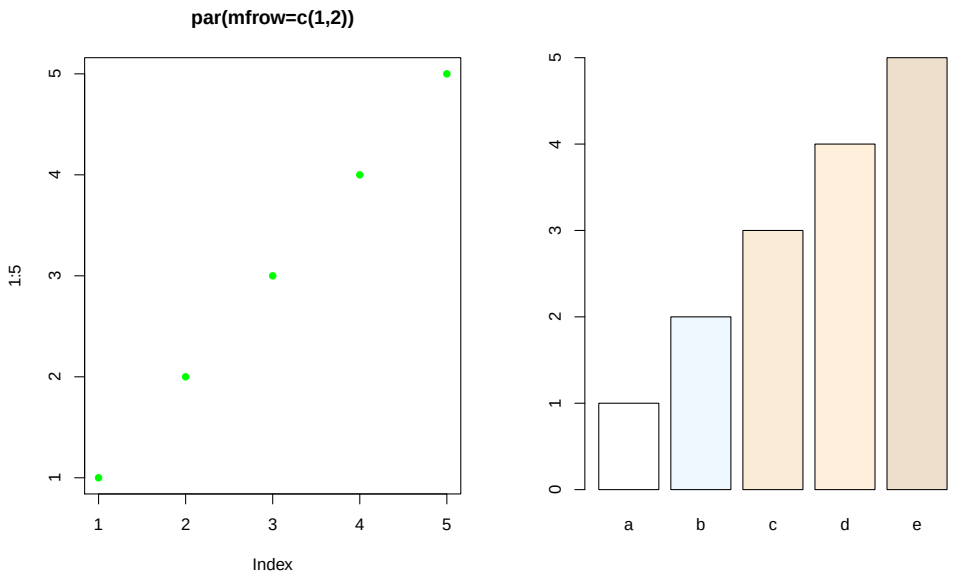


Multiple plots on one page

- change the graphical parameters.
  - `par()` : defines the plot environment
  - change `mfrow` parameter to `par()`
  - `mfrow=c(1,2)` : means create two plots in one row and 2 columns
  - `mfrow=c(2,1)` : means create two plots in two rows and one column

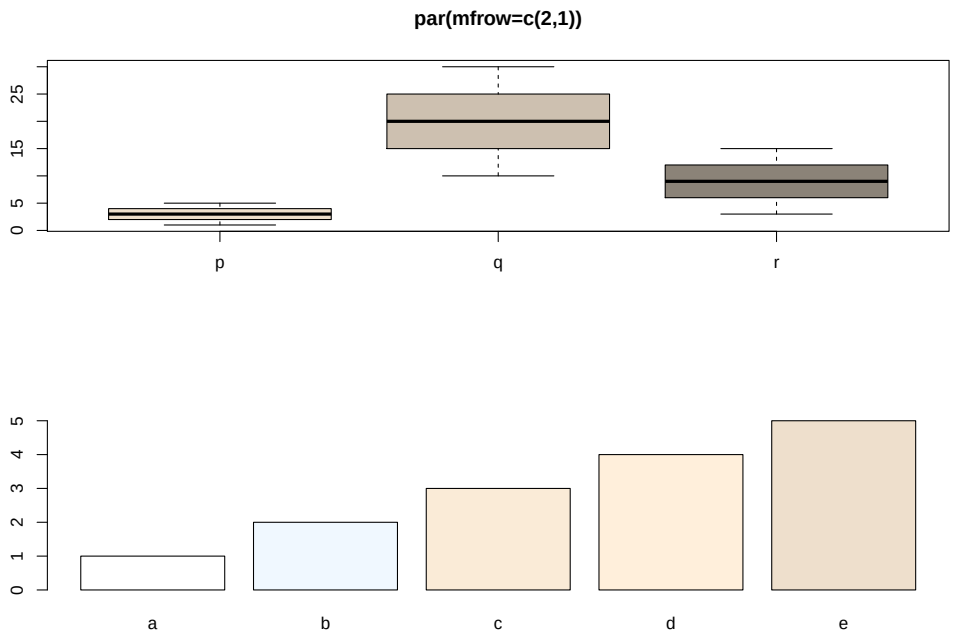
Multiple plots on one page

```
par(mfrow=c(1,2))
plot(1:5, pch=16, col='green',
     main="par(mfrow=c(1,2))")
barplot(height=1:5, names.arg=letters[1:5], col=colors()[1:5])
```



Multiple plots on one page

```
par(mfrow=c(2,1))

boxplot(list(p=1:5,q=10:30, r=3:15 ), pch=1,
        col=colors()[5:10],
        main="par(mfrow=c(2,1))")
barplot(height=1:5, names.arg=letters[1:5], col=colors()[1:5])
```



Save plots

- R has several graphical devices
- most common ones are `pdf`, `png`, `svg`

```
pdf(file="FILE_NAME.pdf", width=11.69, height=8.27)

## define par() here for multiple plots
 plot(1:5)

dev.off()
```

Thank you