# R TUTORIAL

## BUSINESS ANALYTICS (SOSE2013)

Josif Grabocka

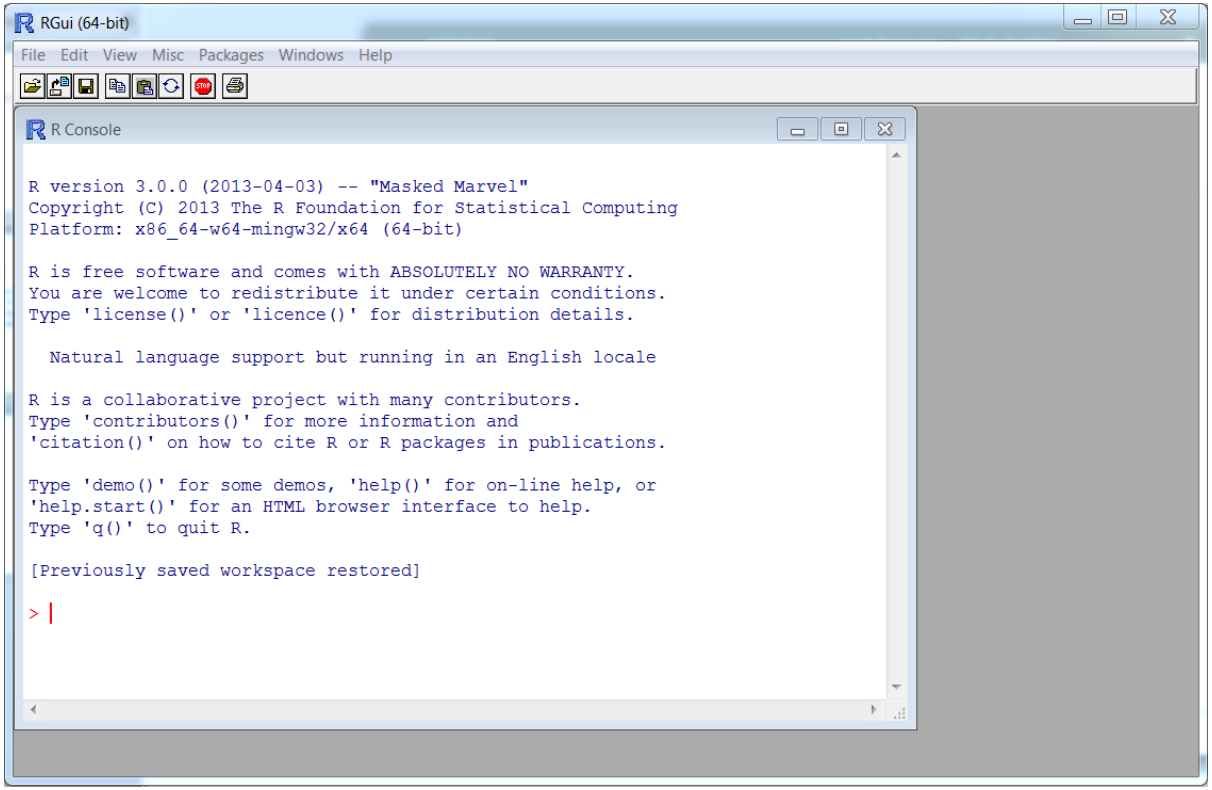
30/04/2013

# What is R?

- Programming language
- Software environment
- Used by Statisticians and Data Miners
- Open Source version of “S”
- Large number of built-in statistical functions
- Easily configurable via packages

# Getting R

- Download it from:
  - http://cran.r-project.org/
- Includes RGui - IDE



# Basics – Math Operations

- Type an expression into the console, while R computes the result

```
> 1+1
[1] 2
> pi
[1] 3.141593
> sqrt(2)
[1] 1.414214
```

# Basics - Variables

- Variables are created and initialized via “name” <- “value”
- A function in R is called as “functionname”()

```
> x <- 23
> y <- 4
> z <- log( sqrt(x) + y )
> print(z)
[1] 2.174278
> z
[1] 2.174278
```

# Basics - Printing

- print() : prints a single variable or data structure
- cat(): prints concatenated content

```
> x <- 5
> y <- 4
> z <- sqrt(x+y)
> print(z)
[1] 3
> cat("Square root of", x, " plus ", y, " is ", z, "\n")
Square root of 5  plus  4  is  3
```

# Basics – Workspace

- The R session workspace stores all the created variables and functions in primary memory (RAM)
- In order to see all the created variables in your workspace use the list command

```
> ls()
[1] "x"  "y"      "z"
```

# Basics – Workspace - Deleting Variables

- Delete through the rm() function

```
> ls()
[1] "x" "y" "z"
> rm(x)
> ls()
[1] "y" "z"
> rm(y,z)
> ls()
character(0)
```

# Basics - Vectors

- A vector is a list of numeric values and is created as
- “name” <- c(“list of numbers”)

```
> v1 <- c(1,4,7,10,13)
> v2 <- c(3,23,2,-1,4)
> mean(v1)
[1] 7
> sd(v2)
[1] 9.576012
> cor(v1,v2)
[1] -0.363252
> cor(v2,v2)
[1] 1
```

# Basics – Comparing Vectors

- Vectors can be compared (like variables) for equality ==, inequality !=, greater > or smaller <
- The outcome is a logical value TRUE/FALSE

```
> v <- c(3, pi, 4)
> w <- c(pi,pi,pi)
> v == w
[1] FALSE  TRUE FALSE
> w > v
[1]  TRUE FALSE FALSE
```

# Basics - Sequences

- Create a sequence of numbers via
- “n”:”m”, for “n”,”n”+1,”n”+2,…,”m”
- seq(“n”,”m”,”k”), for “n”, “n”+”k”, “n”+2”k”, “n”+3”k”, …, “m”

```
> 1:14
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
> seq(-1, 2, 0.3)
 [1] -1.0 -0.7 -0.4 -0.1  0.2  0.5  0.8  1.1  1.4  1.7  2.0
```

# Basics – Selecting Vector Elements (1)

- Use square brackets to access element at a desired position, e.g. v[3] accesses the third element of v
- Use a negative sign to exclude, e.g. v[-2] is all except the second element
- Use a vector of indices to select multiple values
- Use a logical operator to access based on a condition

# Basics – Selecting Vector Elements (2)

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> fib[2]
[1] 1
> fib[7]
[1] 8
> fib[2:5]
[1] 1 1 2 3
> fib[ c(1,3,5,7) ]
[1] 0 1 3 8
> fib[ -(7:10) ]
[1] 0 1 1 2 3 5
```

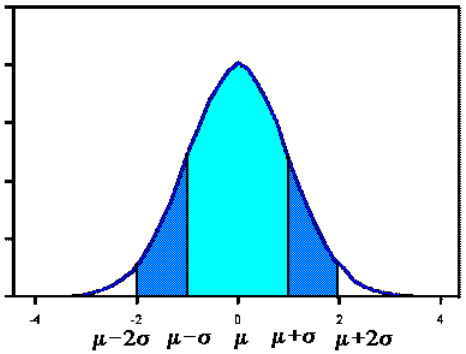
# Basics – Selecting Vector Elements (3)

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> mean(fib)
[1] 8.8
> fib[ fib > mean(fib) ] # fib > mean(fib) vec. of TRUE/FALSE
[1] 13 21 34
> fib[ fib %% 2 == 0 ]
[1]  0  2  8 34
```

# Basics – Vector Arithmetic

```
> v <- c(11,12,13,14,15)
> w <- c(1,2,3,4,5)
> v+w
[1] 12 14 16 18 20
> v/w
[1] 11.000000  6.000000  4.333333  3.500000  3.000000
> v ^ 2
[1] 121 144 169 196 225
> mean(w)
[1] 3
> sd(w)
[1] 1.581139
> (w-mean(w))/sd(w)
[1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
```

Normalization



# Basics - Functions

```
function(param1,param2,…..,paramN)
{
  “expression 1”
  “expression 2”
  …
}
```

```
> gcd <- function(a,b){
+           if(b == 0) return(a)
+           else return( gcd(b, a %% b))
+   }
gcd(10,20)
[1] 10
```

*Conditional Execution*
**if**(“cond”) “expr” **else** “expr”

*Termination*
**return**(“value”)

# Basics - Loops

- **while**(“condition”) “expression”

```
> z <- 0
> while(z < 5){
+    z <- z + 2
+    print(z)
+ }
[1] 2
[1] 4
[1] 6
```

- **Homework: Search for and learn the “for” loop**

# Basics – Getting Help

> help.start() opens a local website on your browser which provides free tutorials, documentation, etc …

For a specific function:

> help(“functionname”)

> args(“functionname”)

> example(“functionname”)

Always ask Google if you get stuck!

# Data Structures - Lists

- List is a vector whose elements are allowed to represent different modes/data types, i.e. numbers, strings, vectors, other lists …
- Lists are created using the **list**(“data”) function

```
> lst <- list(3, "A", 4.5 )
> print(lst)
[[1]]
[1] 3
[[2]]
[1] "A“
[[3]]
[1] 4.5
> length(lst)
[1] 3
> lst[[2]]
[1] "A"
```

```
> mode(lst[[2]])
[1] “character”
> lst2 <- list("Z", c(-21,5,7), list(3,"C"))
> print(lst2)
[[1]]
[1] "Z"
[[2]]
[1] -21   5   7
[[3]]
[[3]][[1]]
[1] 3
[[3]][[2]]
[1] "C"
```

# Data Structures – Remove List Elements

- “listname”[“indices”] <- NULL

```
> lst <- list("A",100,90,"B",80,70,"C",60,50,"D",40,30)
> lst[ seq(2,length(lst),3)] <- NULL
> print(lst)
[[1]]
[1] "A"
[[2]]
[1] 90
[[3]]
[1] "B"
[[4]]
[1] 70
[[5]]
[1] "C"
[[6]]
[1] 50
[[7]]
[1] "D"
[[8]]
[1] 30
```

**length**(“data”): length of vector/list

# Data Structures - Names

- Names can be set to vector and list elements

```
> grades <- c(5,4,3,2,1)
> names(grades) <- c("very bad", "bad", "normal", "good", "very good")
> grades["good"]
good
   2
```

# Data Structures - Matrices

- Matrix is a vector which has two dimensions, set via **dim()**

```
> A <- 1:6
> print(A)
[1] 1 2 3 4 5 6
> dim(A)
NULL
> dim(A) <- c(3,2)
> print(A)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Access: [“row”,”col”]

```
> A[3,1]
[1] 3
> A <- matrix(1:6,3,2)
> print(A)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

(“content”,”rows”,”cols”)

# Data Structure – Matrix Selection

- Select one column/row, or a sub-matrix

```
> A <- matrix(1:16,4,4);
> print(A)
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> A[2,]
[1]  2  6 10 14
> A[,3]
[1]  9 10 11 12
```

```
> A[1:2,3:4]
     [,1] [,2]
[1,]    9   13
[2,]   10   14
```

# Data Structures - Arrays

- N-dimensional data structures

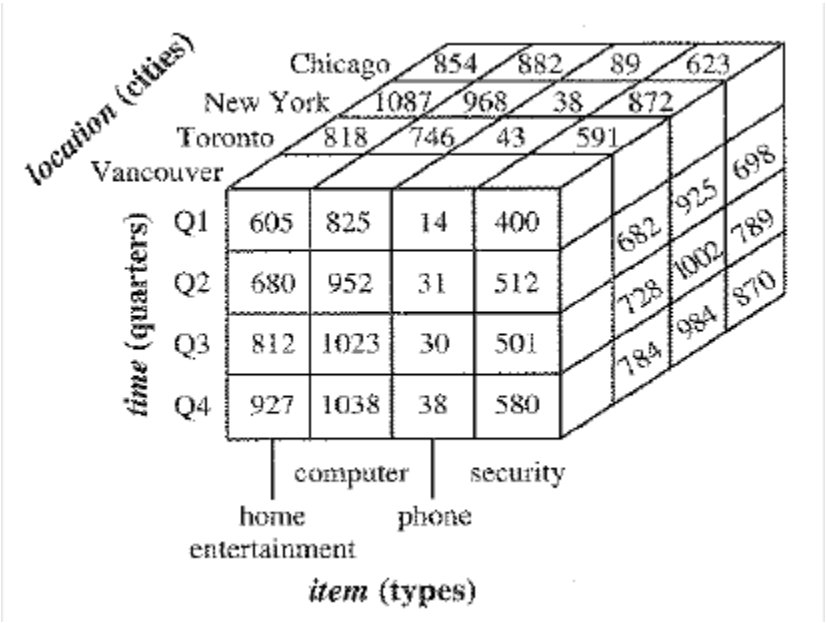
```
> D <- 1:12
> dim(D) <- c(2,3,2)
> print(D)
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Three dimensional data cube, e.g.:




Source: http://timkienthuc.blogspot.de

# Data Structures - Data Frames

- A tabular (2d) data structure which is a list whose elements are vectors. It is created using **data.frame**(“vec1”,”vec2”, …, “vecn”)
- Vectors are columns of the data frame and must have same length.
- For simplicity, think of the data frame like an Excel spreadsheet where each column has a unique data type.

```
> names <- c("hans", "tim", "lukas", "jorg")
> grades <- c(1.7, 2.0, 3.0, 1.3)
> scores <- data.frame(names,grades)
> print(scores)
  names grades
1  hans    1.7
2   tim    2.0
3 lukas    3.0
4  jorg    1.3
```

# Data Structure – Append Data

**Vectors**

```
> v <- c(1,2,3,4)
> v <- c(v,5)
> print(v)
[1] 1 2 3 4 5
> w <- c(6,7,8,9)
> w <- c(v,w)
> print(w)
[1] 1 2 3 4 5 6 7 8 9
```

**Frames**

```
> newRow <- data.frame(
    names="josif", grades=2.0)
> scores <- rbind(scores,newRow)
> print(scores)
 names grades
1  hans    1.7
2  tim     2.0
3  lukas   3.0
4  jorg    1.3
5  josif   2.0
```

# I/O – Read Tabular Files (1)

- Each line one record
- Within a record, each field is delimited by a special character such as comma, space, tab or colon.
- Each record contains the same number of fields

```
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

File “statisticians.txt”

# I/O – Read Tabular Files (2)

- **read.table**(“filepath”) reads the file under the path and returns a data frame with the read content

```
> statisticians <- read.table("statisticians.txt")
> print(statisticians)
      V1        V2          V3    V4
1  Fisher      R.A.        1890  1962
2  Pearson    Karl         1857  1936
3  Cox        Gertrude     1900  1978
4  Yates      Frank        1902  1994
5  Smith      Kirstine     1878  1939
> statisticians$V1
[1] Fisher  Pearson Cox     Yates   Smith
Levels: Cox Fisher Pearson Smith Yates
```

# I/O – Read CSV Files

- Simple CSV file “table.csv” (with a header line)

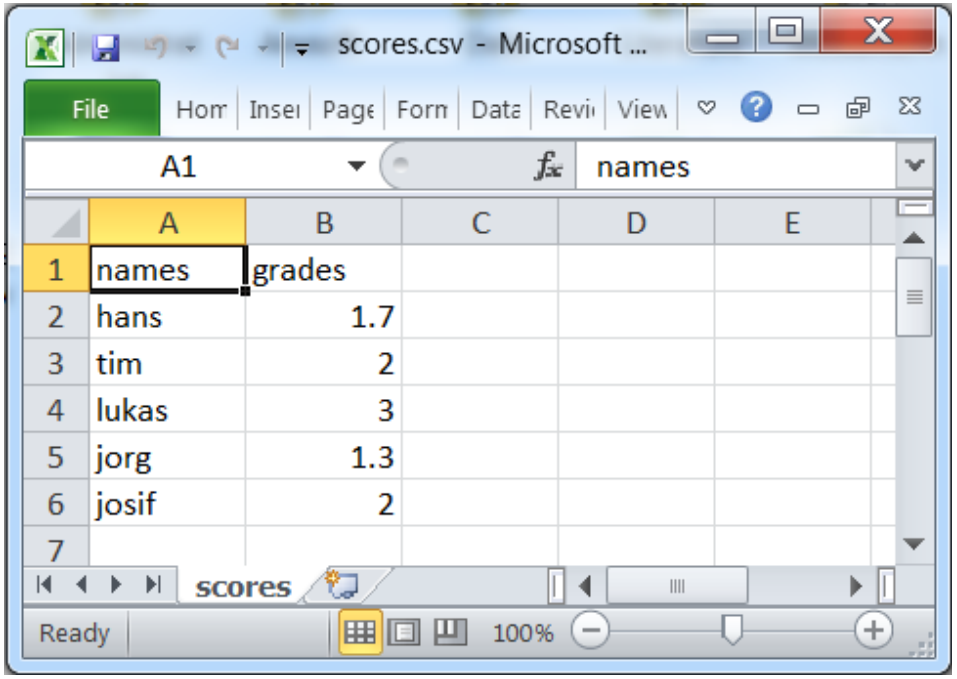
```
label,lbound,ubound
low,0,0.674
mid,0.674,1.64
high,1.64,2.33
```

```
> tbl <- read.csv("table.csv")
> print(tbl)
  label lbound ubound
1   low  0.000  0.674
2   mid  0.674  1.640
3  high  1.640  2.330
```

# I/O – Write Data Frame to CSV File

```
> print(scores)
  names grades
1  hans    1.7
2   tim    2.0
3 lukas    3.0
4  jorg    1.3
5 josif    2.0
```

```
> write.csv(scores, "scores.csv",
                    row.names=F)
```



# Strings

```
> name <- "josif"
> surname <- "grabocka"
> fullname <- paste(name,surname)
> print(fullname)
[1] "josif grabocka"
> nchar(fullname)
[1] 14
> substr(fullname,7,10)
[1] "grab“
> sub("a", "$", fullname)
[1] "josif gr$bocka"
> gsub("a", "$", fullname)
[1] "josif gr$bock$"
```

# Dates

```
> Sys.Date()
[1] "2013-04-28"
> format(Sys.Date(), "%m/%d/%Y")
[1] "04/28/2013“

> s <- as.Date("2013-04-23")
> e <- as.Date("2013-04-30")
> seq(s,e,1)
[1] "2013-04-23" "2013-04-24" "2013-04-25" "2013-04-26"
"2013-04-27" "2013-04-28" "2013-04-29" "2013-04-30"
```

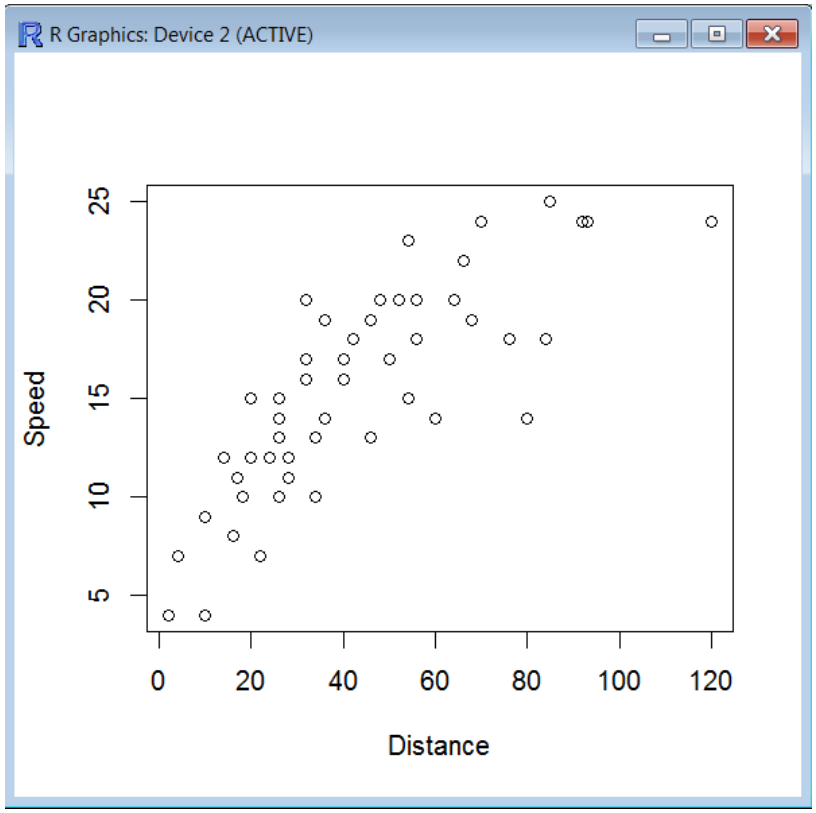
# Graphics - Introduction

- Creating plots, charts and visual presentation of results
- High-level graphics functions
  - **plot** – generic plotting
  - **boxplot** – a box plot
  - **histogram** – histogram visualization of data
  - **curve** – display a function
- Low-level graphics functions (inside high-level containers)
  - **lines** – add lines to the plot
  - **points** – add points to the high level function
  - **polygon** – addition of polygon data
  - **text** – insertion of text annotation inside plot
- Title, Legend, Colors, Line-Styles, etc …

# Graphics – Scatter Plot

- Packages :: Load Package :: Datasets (default datasets)

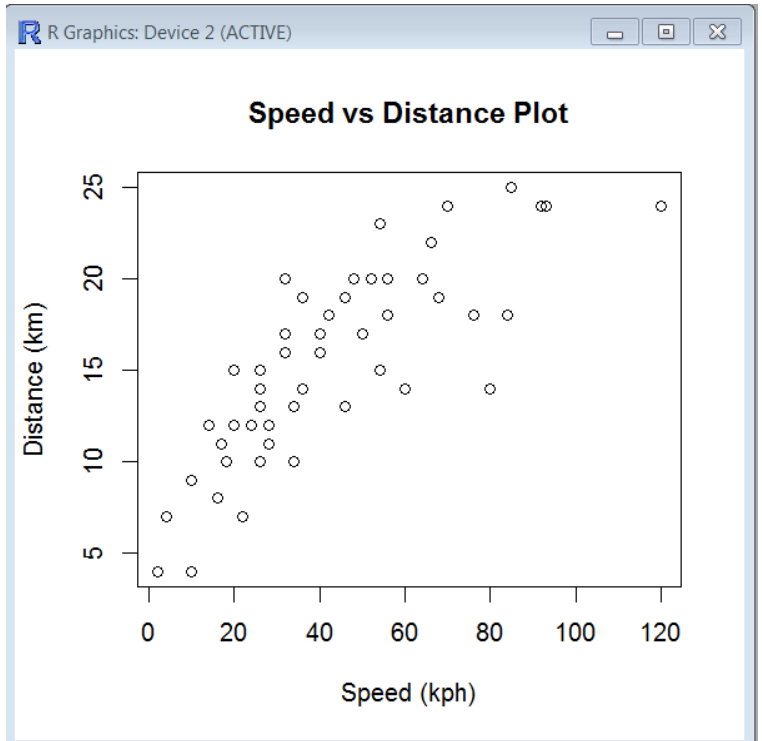
```
> ds <- cars
> print(ds)
speed dist
1     4    2
2     4   10
…
50    25   85
> Distance <- ds$dist #ds[,1]
> Speed <- ds$speed #ds[,2]
> plot(Distance,Speed)
```



# Graphics – Title and Axis Labels

*plot(D, main="title", xlab="X-axis label", ylab="Y-axis label")*

> plot(Distance,Speed, main="Speed vs Distance Plot", xlab="Speed (kph)", ylab="Distance (km)")



# Graphics – Multiple Plots (1)

- Set the parameter **mfrow** before calling plot
- *par(mfrow=c("numRows","numCols"))*

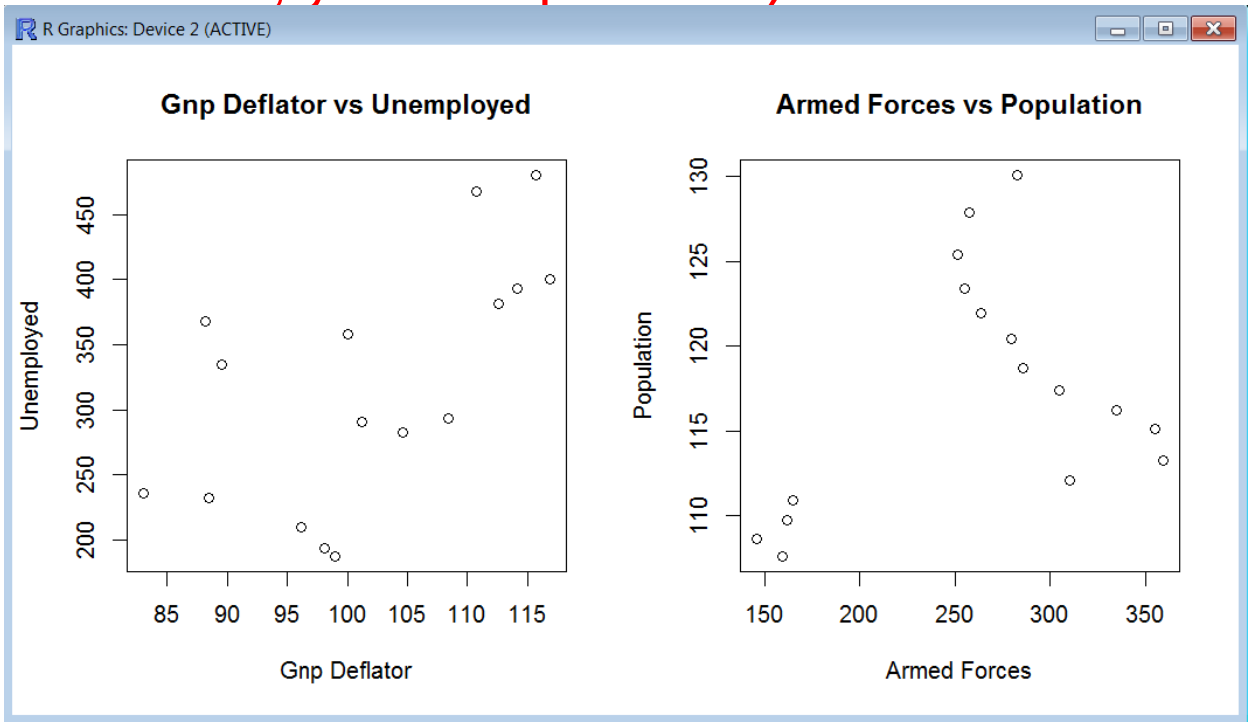
```
> ds <- longley
> print(ds)
```

| | GNP.deflator | GNP | Unemployed | Armed.Forces | Population | Year | Employed |
|---|---|---|---|---|---|---|---|
| 1947 | 83.0 | 234.289 | 235.6 | 159.0 | 107.608 | 1947 | 60.323 |
| 1948 | 88.5 | 259.426 | 232.5 | 145.6 | 108.632 | 1948 | 61.122 |
| 1949 | 88.2 | 258.054 | 368.2 | 161.6 | 109.773 | 1949 | 60.171 |
| 1950 | 89.5 | 284.599 | 335.1 | 165.0 | 110.929 | 1950 | 61.187 |
| 1951 | 96.2 | 328.975 | 209.9 | 309.9 | 112.075 | 1951 | 63.221 |
| 1952 | 98.1 | 346.999 | 193.2 | 359.4 | 113.270 | 1952 | 63.639 |
| 1953 | 99.0 | 365.385 | 187.0 | 354.7 | 115.094 | 1953 | 64.989 |
| 1954 | 100.0 | 363.112 | 357.8 | 335.0 | 116.219 | 1954 | 63.761 |
| 1955 | 101.2 | 397.469 | 290.4 | 304.8 | 117.388 | 1955 | 66.019 |
| 1956 | 104.6 | 419.180 | 282.2 | 285.7 | 118.734 | 1956 | 67.857 |
| 1957 | 108.4 | 442.769 | 293.6 | 279.8 | 120.445 | 1957 | 68.169 |
| 1958 | 110.8 | 444.546 | 468.1 | 263.7 | 121.950 | 1958 | 66.513 |
| 1959 | 112.6 | 482.704 | 381.3 | 255.2 | 123.366 | 1959 | 68.655 |
| 1960 | 114.2 | 502.601 | 393.1 | 251.4 | 125.368 | 1960 | 69.564 |
| 1961 | 115.7 | 518.173 | 480.6 | 257.2 | 127.852 | 1961 | 69.331 |
| 1962 | 116.9 | 554.894 | 400.7 | 282.7 | 130.081 | 1962 | 70.551 |

**Plot together "GNP.Deflator (col 1) vs Unemployed (col 2)" and "Armed. (col 4) vs Pop. (col 5)"?**

# Graphics – Multiple Plots (2)

```
> par(mfrow=c(1,2))
> plot(ds[,1],ds[,3], main="Gnp Deflator vs Unemployed", xlab="Gnp Deflator", ylab="Unemployed")
> plot(ds[,4],ds[,5], main="Armed Forces vs Population", xlab="Armed Forces", ylab="Population")
```



# Data Transformations – lapply, sapply

- Apply a “function” to every element of a “list” or vector
- R strategy against iterations!

```
lst <- lapply(“list”, “function”)          # output is a list
vec <- sapply(“list”, “function”)          # output is a vector
```

```
> funStuff <- c("Beer", "Football", "Love", "Statistics")
> greatify <- function(str){ return ( paste(str,"is great!") ) }
> funStuffGreatified <- sapply(funStuff, greatify)
> print(funStuffGreatified)
```

| Beer | Football | Love | Statistics |
|---|---|---|---|
| "Beer is great!" | "Football is great!" | "Love is great!" | "Statistics is great!" |

# Data Transformation – Matrix apply (1)

- apply("matrix", "1-row or 2-column", "function")
- Example: Compute the product of each row?

```
> M <- matrix(1:16, 4, 4)
> print(M)
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> apply(M, 1, prod)
[1]  585 1680 3465 6144
> 1*5*9*13
[1] 585
```

# Data Transformation – Matrix apply (2)

- Compute the factorial of each cell of a matrix!
- “for every column -> for every row element of that column”?

```
> M <- matrix(1:9, 3, 3)
> print(M)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> factorial <- function(x) { if(x < 1) return(1) else return( x*factorial(x-1) ) }
> apply(M, 2, function(row){ return( sapply(row,factorial) ) })
     [,1] [,2]   [,3]
[1,]    1   24   5040
[2,]    2  120  40320
[3,]    6  720 362880
```

# Data Transformation – Data Frames

- By default a function is applied to columns, which are vectors
- lst <- **lapply**(“matrix”, “function”) # output is list
- vec <- **sapply**(“matrix”, “function”) # output is vector
- Also **apply** can be used for data frames similar to matrices, however the data frame columns must have identical modes/data types

THE END