## Statistical Data Analysis

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr

http://www3.yildiz.edu.tr/~naydin

# Introduction to R

## The R Project

- Environment for statistical computing and graphics
- Free software
- Associated with simple programming language
  - Similar to S and S-plus
  - www.r-project.org

## R, S and S-plus

- S is an interactive environment for data analysis developed at Bell Laboratories since 1976
  - 1988 - S2: RA Becker, JM Chambers, A Wilks
  - 1992 - S3: JM Chambers, TJ Hastie
  - 1998 - S4: JM Chambers
- Exclusively licensed by AT&T/Lucent to Insightful Corporation, Seattle WA.
  - Product name is “S-plus”.
- Implementation languages C, Fortran.

## R, S and S-plus

- R is initially written by Ross Ihaka and Robert Gentleman at Dep. of Statistics of University of Auckland, New Zealand during 1990s.
- GNU General Public License (GPL)
  - can be used by anyone for any purpose
- Open Source
  - efficient bug tracking and fixing system supported by the user community

– http://cm.bell-labs.com/cm/ms/departments/sia/S/history.html

## Compiled C vs Interpreted R

- C requires a complete program to run
  - Program is translated into machine code
  - Can then be executed repeatedly
- R can run interactively
  - Statements converted to machine instructions as they are encountered
  - This is much more flexible, but also slower
- R Programming Language
  - Interpreted language

# R and statistics

- Packaging:
  - a crucial infrastructure to efficiently produce, load and keep consistent software libraries from (many) different sources / authors
- Statistics:
  - most packages deal with statistics and data analysis
- State of the art:
  - many statistical researchers provide their methods as R packages

# Tutorials

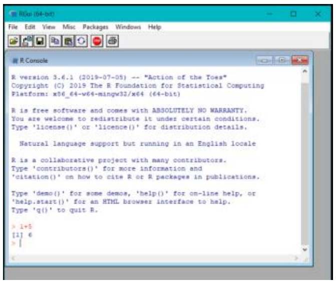
- From R website under “Documentation”
  - “Manual” is the listing of official R documentation
    - An Introduction to R
    - R Language Definition
    - Writing R Extensions
    - R Data Import/Export
    - R Installation and Administration
    - The R Reference Index

# Tutorials

- “Contributed” documentation are tutorials and manuals created by R users
  - Simple R
  - R for Beginners
  - Practical Regression and ANOVA Using R
- R FAQ
- Mailing Lists (listserv)
  - r-help

# Interactive R

- R defaults to an interactive mode



```
R version 3.6.1 (2019-07-05) -- "Action of the Toes"
Copyright (C) 2019 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 1+5
[1] 6
> 
```

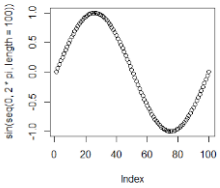
- A prompt “>” is presented to users
- Each input expression is evaluated…
- … and a result returned

# R as a Calculator

```
> 1 + 1          # Simple Arithmetic
[1]  2
> 2 + 3 * 4      # Operator precedence
[1]  14
> 3 ^ 2          # Exponentiation
[1]  9
> exp(1)         # Basic mathematical functions are available
[1]  2.718282
> sqrt(10)
[1]  3.162278
> pi             # The constant pi is predefined
[1]  3.141593
> 2*pi*6378      # Circumference of earth at equator (in km)
[1]  40074.16
```

# R as a Calculator

```
> log2(32)
[1]  5
> seq(0, 5, length=6)
[1]  0 1 2 3 4 5
```



```
> plot(sin(seq(0, 2*pi, length=100)))
```

## Variables in R

- Numeric
  - Store floating point values
    ```
    > a = 49
    ```
- Boolean (T or F)
  - Values corresponding to True or False
    ```
    > a = (1+1==3)
    > a
    [1] FALSE
    ```
- Strings
  - Sequences of characters
    ```
    a = "The dog ate my homework"
    > sub("dog","cat",a)
    [1] "The cat ate my homework“
    ```
- Type determined automatically when variable is created with "<-" operator

## R as a Smart Calculator

```
> x <- 1                    # Can define variables
> y <- 3                    # using "<-" operator to set values
> z <- 4
> x * y * z
[1] 12

> X * Y * Z                 # Variable names are case sensitive
Error: Object "X" not found

> This.Year <- 2004     # Variable names can include period
> This.Year
[1] 2004
```

## Missing Values

- Variables of each data type (numeric, character, logical) can also take the value NA: not available.
  - NA is not the same as 0
  - NA is not the same as ""
  - NA is not the same as FALSE
- Any operations (calculations, comparisons) that involve NA may or may not produce NA:
  ```
  > NA==1
  [1] NA
  > 1+NA
  [1] NA
  > max(c(NA, 4, 7))
  [1] NA
  > max(c(NA, 4, 7), na.rm=T)
  [1] 7
  > NA | TRUE
  [1] TRUE
  > NA & TRUE
  ```

## Functions and Operators

- Functions do things with data
  - “Input”: function arguments (0,1,2,…)
  - “Output”: function result (exactly one)
  - Example:
    ```
    add = function(a,b)
    { result = a+b
    return(result) }
    ```
- Operators:
  - Short-cut writing for frequently used functions of one or two arguments.
    - Examples: + - * / ! & | %%

## R Vectors

- An ordered collection of data of the same type
  - Created with
    - c() to concatenate elements or sub-vectors
      ```
      > a = c(1,2,3)
      > a*2
      [1]  2 4 6
      ```
    - rep() to repeat elements or patterns
    - seq() or m:n to generate sequences
- Most mathematical functions and operators can be applied to vectors
  - Without loops!

## Defining Vectors

```
> rep(1,10)             # repeats the number 1, 10 times
[1]  1 1 1 1 1 1 1 1 1 1
> seq(2,6)              # sequence of integers between 2 and 6
[1]  2 3 4 5 6          # equivalent to 2:6
> seq(4,20,by=4)        # Every 4th integer between 4 and 20
[1]  4 8 12 16 20
> x <- c(2,0,0,4)       # Creates vector with elements 2,0,0,4
> y <- c(1,9,9,9)
> x + y                 # Sums elements of two vectors
[1]  3 9 9 13
> x * 4                  # Multiplies elements
[1]  8 0 0 16
> sqrt(x)                        # Function applies to each element
[1]  1.41 0.00 0.00 2.00         # Returns vector
```

## Accessing Vector Elements

- Use the [ ] operator to select elements
- To select specific elements:
  - Use index or vector of indexes to identify them
- To exclude specific elements:
  - Negate index or vector of indexes
- Alternative:
  - Use vector of T and F values to select subset of elements

## Accessing Vector Elements

```
> x <- c(2,0,0,4)
> x[1]                  # Select the first element, equivalent to x[c(1)]
[1] 2
> x[-1]                 # Exclude the first element
[1] 0 0 4
> x[1] <- 3 ; x
[1] 3 0 0 4
> x[-1] = 5 ; x
[1] 3 5 5 5
> y < 9                 # Compares each element, returns result as vector
[1] TRUE FALSE FALSE FALSE
> y[4] = 1
> y < 9
[1] TRUE FALSE FALSE TRUE
> y[y<9] = 2            # Edits elements marked as TRUE in index vector
> y
[1] 2 9 9 2
```

## Matrices and Arrays

- matrix: a rectangular table of data of the same type
  - example:
    - the expression values for 10000 genes for 30 tissue biopsies: a matrix with 10000 rows and 30 columns.
- array: 3-,4-,..dimensional matrix
  - example:
    - the red and green foreground and background values for 20000 spots on 120 chips: a 4 x 20000 x 120 (3D) array.

## Lists

- vector:
  - an ordered collection of data of the same type.
    ```
    > a = c(7,5,1)
    > a[2]
    [1] 5
    ```
- list:
  - an ordered collection of data of arbitrary types.
    ```
    > x = list(ad="ali", yas=30, bekar=F)
    ```
- Typically, vector elements are accessed by their index (an integer), list elements by their name (a character string).
  - But both types support both access methods.
  - the following all retrieve ad:
    ```
    > x$ad          > x["ad"]       > x[1]          > x[-2:-3]
    [1] "ali"        [1] "ali"       [1] "ali"       [1] "ali"
    ```

## Data Frames

- Group a collection of related vectors
  - Most of the time, when data is loaded, it will be organized in a data frame
    - It is a rectangular table with rows and columns;
      - data within each column has the same type (e.g. number, text, logical), but different columns may have different types.
- Example:

```
> a
        localization tumorsize  progress
XX348     proximal         6.3    FALSE
XX234        distal        8.0     TRUE
XX987     proximal        10.0    FALSE
```

## Setting Up Data Sets

- Load from a text file using read.table()
  - Parameters header, sep, and na.strings control useful options
  - read.csv() and read.delim() have useful defaults for comma or tab delimited files
- Create from scratch using data.frame()
  - Example:

    data.frame(height=c(150,160), weight=(65,72))

## Blood Pressure Data Set

| HEIGHT | WEIGHT | WAIST | HIP | BPSYS | BPDIA |
|---|---|---|---|---|---|
| 172 | 72 | 87 | 94 1 | 27.5 | 80 |
| 166 | 91 | 109 | 107 | 172.5 | 100 |
| 174 | 80 | 95 | 101 | 123 | 64 |
| 176 | 79 | 93 | 100 | 117 | 76 |
| 166 | 55 | 70 | 94 | 100 | 60 |
| 163 | 76 | 96 | 99 | 160 | 87.5 |

...

- Read into R using:

```
bp <- read.table(“bp.txt”, header=T, na.strings=c(“x”))
```

## Accessing Data Frames

- Multiple ways to retrieve columns…
- The following all retrieve weight data:

```
> bp[“WEIGHT”]
> bp[,2]
> bp$WEIGHT
```

- The following excludes weight data:

```
> bp[,-2]
```

## Factors

- A character string can contain arbitrary text.
- Sometimes it is useful to use a limited vocabulary, with a small number of allowed words.
- A factor is a variable that can only take such a limited number of values, which are called levels.

```
> a
[1] Kolon(Rektum)   Magen               Magen
[4] Magen           Magen               Retroperitoneal
[7] Magen           Magen(retrogastral) Magen
Levels:  Kolon(Rektum)  Magen   Magen(retrogastral)
Retroperitoneal
```

## Factors

```
> class(a)
[1] "factor"
> as.character(a)
[1] "Kolon(Rektum)"  "Magen"                 "Magen"
[4] "Magen"          "Magen"                 "Retroperitoneal"
[7] "Magen"          "Magen(retrogastral)"   "Magen"
> as.integer(a)
[1] 1 2 2 2 2 4 2 3 2
> as.integer(as.character(a))
[1] NA NA NA NA NA NA NA NA NA NA NA NA
Warning message:
NAs introduced by coercion
```

## Subsetting

- Individual elements of a vector, matrix, array or data frame are accessed with "[ ]" by specifying their index, or their name

```
> a
            localisation    tumorsize    progress
XX348       proximal        6.3          0
XX234       distal          8.0          1
XX987       proximal        10.0         0

> a[3, 2]
[1] 10

> a["XX987", "tumorsize"]
[1] 10

> a["XX987",]
            localisation    tumorsize    progress
XX987       proximal        10           0
```

## Subsetting

```
> a
            localisation    tumorsize    progress
XX348       proximal        6.3          0
XX234       distal          8.0          1
XX987       proximal        10.0         0
```

- subset rows by a vector of indices

```
> a[c(1,3),]
            localisation    tumorsize    progress
XX348       proximal        6.3          0
XX987       proximal        10.0         0
```

- subset rows by a logical vector

```
> a[c(T,F,T),]
            localisation    tumorsize    progress
XX348       proximal        6.3          0
XX987       proximal        10.0         0
```

## Subsetting

```
> a
                localisation    tumorsize     progress
XX348           proximal        6.3           0
XX234           distal          8.0           1
XX987           proximal        10.0          0
```

- subset a column

```
> a$localisation
[1] "proximal"     "distal"   "proximal"
```

- comparison resulting in logical vector

```
>  a$localisation=="proximal"
[1]  TRUE FALSE  TRUE
```

- subset the selected rows

```
> a[ a$localisation=="proximal", ]
                localisation    tumorsize     progress
XX348           proximal        6.3           0
XX987           proximal        10.0          0
```

## Common Forms of Data in R

- Variables are created as needed

- Numeric values
- Vectors
- Data Frames
- Lists

- Used some simple functions:
  - c(), seq(), read.table(), …

## Programming Constructs

- Grouped Expressions
- Control statements
  - if … else …

  - for loops
  - repeat loops
  - while loops

  - next, break statements

## Grouped Expressions

{expr_1; expr_2; … }

- Valid wherever single expression could be used

- Return the result of last expression evaluated

- Relatively similar to compound statements in C

## Branching (if … else …)

if (expr_1) expr_2 else expr_3

- The first expression should return a single logical value
  - Operators && or || may be used
- Conditional execution of code

```
if (logical expression)
{
        statements
}
else
{
        alternative statements
}

else branch is optional
```

## Example: if … else …

```
# Standardize observation i
if (sx[i] == "male")
  {
  z[i] <- (obsrvd[i] - males.mean) / males.sd;
  }
else
{
  z[i] <- (obsrvd[i] - females.mean) / females.sd;
```

## Loops (for)

- When the same or similar tasks need to be performed multiple times; for all elements of a list; for all columns of an array; etc.

```
for (name in expr_1) expr_2
```

  - name is the loop variable
  - expr_1 is often a sequence
    - e.g. 1:20
    - e.g. seq(1, 20, by = 2)

## Example: for

```
# Sample M random pairings in a set of N objects
for (i in 1:M)
        {
        # As shown, the sample function returns a
        single
                # element in the interval 1:N
                p = sample(N, 1)
                q = sample(N, 1)
        # Additional processing as needed…
        ProcessPair(p, q);
        }
```

## repeat

```
repeat expr
```

- Continually evaluate expression
- Loop must be terminated with break statement

## Example: repeat

```
# Sample with replacement from a set of N objects
# until the number 615 is sampled twice
M <- matches <- 0
repeat
        {
        # Keep track of total connections sampled
        M <- M + 1
        # Sample a new connection
        p = sample(N, 1)
        # Increment matches whenever we sample 615
        if (p == 615)
                matches <- matches + 1;
        # Stop after 2 matches
        if (matches == 2)
                break;
        }
```

## while

```
while (expr_1) expr_2
```

- While expr_1 is false, repeatedly evaluate expr_2
- break and next statements can be used within the loop

## Example: while

```
# Sample with replacement from a set of N objects
# until 615 and 815 are sampled consecutively
match <- false
while (match == false)
        {
        # sample a new element
        p = sample(N, 1)
        # if not 615, then goto next iteration
        if (p != 615)
                next;
        # Sample another element
        q = sample(N, 1)
        # Check if we are done
        if (q != 815)
                match = true;
        }
```

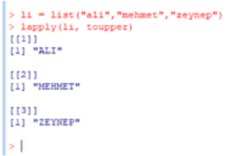
## Example: for and while

```
for (i in 1:10)
{
        print(i*i)
}

i=1
while (i<=10)
{
        print(i*i)
        i = i+1
}
```

## lapply, sapply, apply

- When the same or similar tasks need to be performed multiple times for all elements of a list or for all columns of an array.
  - May be easier and faster than “for” loops
- lapply( li, fct )
  - To each element of the list li, the function fct is applied.
  - The result is a list whose elements are the individual fct results.

```
> li = list("ali","mehmet","zeynep")
> lapply(li, toupper)
[[1]]
[1] "ALİ"
[[2]]
[1] "MEHMET"
[[3]]
[1] "ZEYNEP"
```

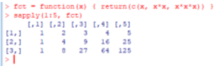


```
> li = list("ali","mehmet","zeynep")
> lapply(li, toupper)
[[1]]
[1] "ALI"

[[2]]
[1] "MEHMET"

[[3]]
[1] "ZEYNEP"

> |
```

## lapply, sapply, apply

- sapply( li, fct )
  - Like lapply, but tries to simplify the result, by converting it into a vector or array of appropriate size

```
> li = list("ali","mehmet","zeynep")
> sapply(li, toupper)
[1] "ALİ"        "MEHMET"     "ZEYNEP«

> fct = function(x) { return(c(x, x*x, x*x*x)) }
> sapply(1:5, fct)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
```



```
> fct = function(x) { return(c(x, x*x, x*x*x)) }
> sapply(1:5, fct)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    4    9   16   25
[3,]    1    8   27   64  125
> |
```

## lapply, sapply, apply

- apply( arr, margin, fct )
  - Applies the function fct along some dimensions of the array arr, according to margin, and returns a vector or array of the appropriate size.

```
> x
     [,1] [,2] [,3]
[1,]    5    7    0
[2,]    7    9    8
[3,]    4    6    7
[4,]    6    3    5

> apply(x, 1, sum)
[1] 12 24 17 14

> apply(x, 2, sum)
[1] 22 25 20
```

## Functions in R

- Easy to create your own functions in R
  - As tasks become complex, it is a good idea to organize code into functions that perform defined tasks
    - In R, it is good practice to give default values to function arguments
- Functions can be defined as

```
        name <- function(arg1, arg2, …)
                expression
```

- Arguments can be assigned default values:

```
        arg_name = expression
```

- Return value is the last evaluated expression or can be set explicitly with return()

## Defining Functions

```
> square <- function(x = 10) x * x
> square()
[1] 100
> square(2)
[1] 4
> intsum <- function(from=1, to=10)
        {
        sum <- 0
        for (i in from:to)
                sum <- sum + i
        sum
        }
> intsum(3)              # Evaluates sum from 3 to 10 …
[1] 52
> intsum(to = 3)         # Evaluates sum from 1 to 3 …
[1] 6
```

## Some notes on functions …

- You can print the arguments for a function using args() command

```
> args(intsum)
function (from = 1, to = 10)
```

- You can print the contents of a function by typing only its name, without the ()
- You can edit a function using

```
> my.func <- edit(my.old.func)
```

## Debugging Functions

- Toggle debugging for a function with debug() / undebug() command
- With debugging enabled, R steps through function line by line
  - Use print() to inspect variables along the way
  - Press <enter> to proceed to next line

```
> debug(intsum)
> intsum(10)
```

## Useful R Functions - Random Generation

- In contrast to many C implementations, R generates pretty good random numbers
- set.seed(seed) can be used to select a specific sequence of random numbers
- sample(x, size, replace = FALSE) generates a sample of size elements from x.
  - If x is a single number, sample is from 1:x

## Useful R Functions - Random Generation

- Samples from Uniform distribution
  - runif(n, min = 1, max = 1)
- Samples from Binomial distribution
  - rbinom(n, size, prob)
- Samples from Normal distribution
  - rnorm(n, mean = 0, sd = 1)
- Samples from Exponential distribution
  - rexp(n, rate = 1)
- Samples from T-distribution
  - rt(n, df)
- And others!

## R Help System

- R has a built-in help system with useful information and examples
  - help() provides general help
  - help(plot) will explain the plot function
  - help.search("histogram") will search for topics that include the word histogram
- example(plot) will provide examples for the plot function

## Input / Output

- Use sink(file) to redirect output to a file
- Use sink() to restore screen output
- Use print() or cat() to generate output inside functions
- Use source(file) to read input from a file

## Basic Utility Functions

- length() returns the number of elements
- mean() returns the sample mean
- median() returns the sample median
- range() returns the largest and smallest values
- unique() removes duplicate elements
- summary() calculates descriptive statistics
- diff() takes difference between consecutive elements
- rev() reverses elements

## Managing Workspaces

- As you generate functions and variables, these are added to your current workspace
- Use ls() to list workspace contents
- Use rm() to delete variables or functions
- When you quit, with the q() function, you can save the current workspace for later use