R - Programming in Biosciences

Dr. Timokratis Karamitros
Senior Researcher

Head, Unit of Bioinformatics and Applied Genomics
Hellenic Pasteur Institute, Athens, GR

tkaram@pasteur.gr

What is R?

R is a dialect of the S language
S is a language that was developed by John Chambers and others at Bell Labs.
S was initiated in 1976 as an internal statistical analysis environment
Version 4 of the S language was released in 1998 and is the version we use today.
The fundamentals of the S language itself has not changed dramatically since 1998.

1991: R was Created in New Zealand by Ross Ihaka and Robert Gentleman.

What is R?

1993: First announcement of R to the public.
1995: Martin M¨achler convinces Ross and Robert to use the GNU General Public License to make R **free** software.

1996: A public mailing list is created (R-help and R-devel)
1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
2000: R version 1.0.0 is released.
2020: R version 4.0.0 is released on April 2020

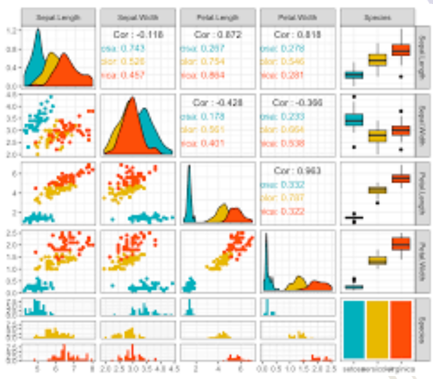
Features of R

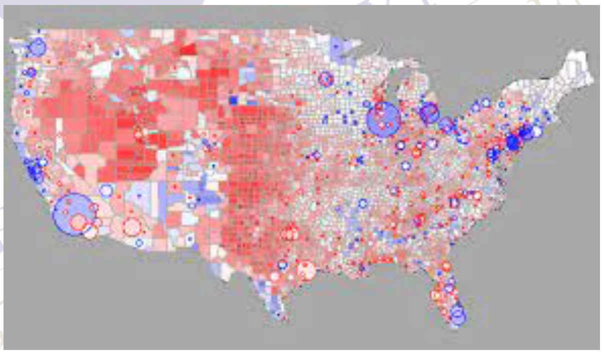
Syntax is similar to S

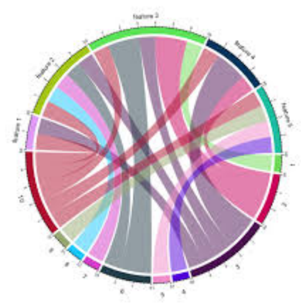
Runs on all computing platform and operational systems

Active development with frequent releases

Graphics capabilities are better than most other statstical packages.







Features of R

The user becomes a programmer since it contains a programming language making the the development of new tools easy

Very active user community; R-help and R-devel mailing lists, Stack Overflow

It is free!

Features of R

With free software, you have the freedom:

To run the program, for any purpose (freedom 0).

To study how the program works, and adapt it to your needs (freedom 1).

To redistribute copies so you can help others (freedom 2).

To improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3).

Access to the source code is a precondition for the above

Drawbacks

R is based on 40 year old technology.

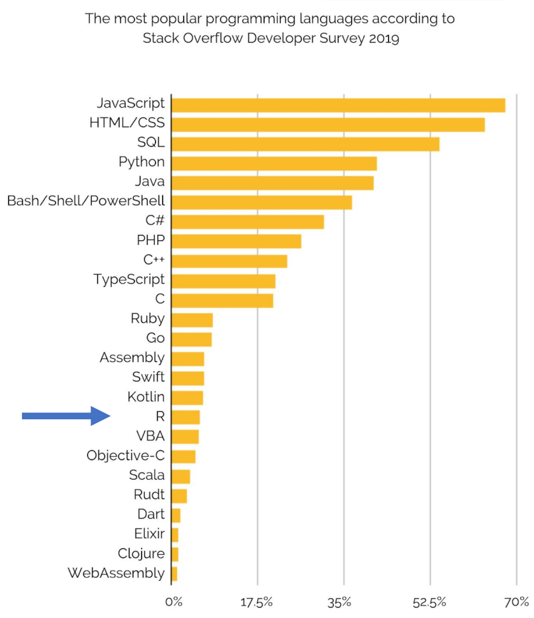
Functionality is based on user contributions.
You have to develop your methods if nobody has done this before

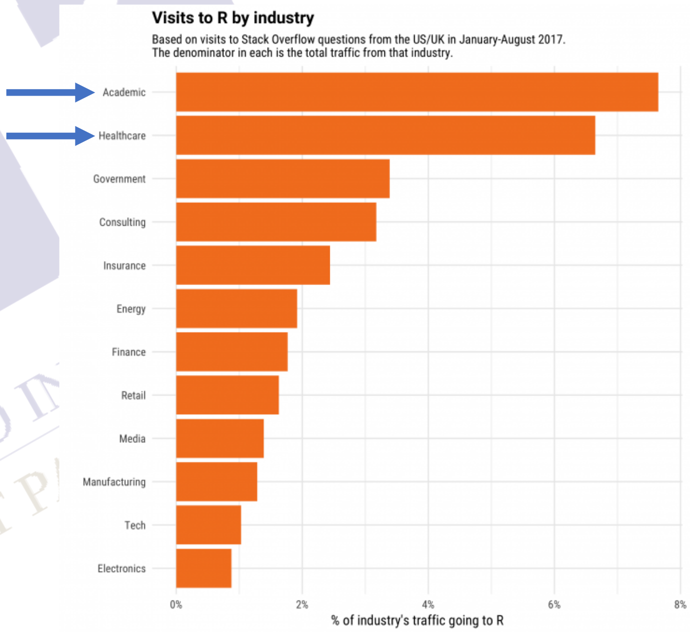
Objects must generally be stored in physical memory,
calculations are performed by a single core, most of the times.

Like in all software packages it is not ideal for all possible situations

Popularity

Although not a famous programming language, is considered an academic standard





Design of the R System

The R programming language is divided into 2 major parts:
1. The "base" R system that we download from CRAN (http://cran.r-project.org)
2. Everything else, usually packages, datasets etc

The "base" R system contains the base package which is required to run R and contains the most fundamental functions.

Design of the R System

Thousands of packages available:

More than 4000 packages on CRAN have been developed by users and programmers around the world.

Importantly… the Bioconductor project (http://bioconductor.org).

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data.
Bioconductor is also open source and open development.
It has two releases each year, and an active user community.
Bioconductor is also available as an AMI (Amazon Machine Image) and Docker images.

People often make packages available on their personal websites;
there is no reliable way to keep track of how many packages are available in this fashion.

R resources

Available from CRAN (http://cran.r-project.org):
An Introduction to R
Writing R Extensions
R Data Import/Export
R Installation and Administration

Books and more
Chambers (2008). Software for Data Analysis, Springer.
Murrell (2005). R Graphics, Chapman & Hall/CRC Press.
Springer series of books called Use R!.
And many more… http://www.r-project.org/doc/bib/R-books.html

LET'S GO

```
Start<-function(willingness)
Development<-function(hard.work)
```

Wait… what if I get stacked???

Where and how I can get help?

- Try to find an answer by searching the Web (Google is your friend).
- Try to find an answer by reading the manual.
- Try to find an answer by inspection or experimentation → make toy datasets, check yourself
- Try to find an answer by asking a skilled friend.
- It's important to let other people know that you've done all of the previous things already
- What steps will reproduce the problem?
- What is the expected output? What do you see instead?
- What version of software (e.g. R, packages, etc.) are you using?
- Which operating system? Hardware limitations?
- Additional information

Data Types and Basic Operations

Objects

R has five basic or "atomic" **classes** of objects:
- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

Handy: **class()** function

The most basic **object** is a **vector**

A vector can only contain objects of the same class

Empty vectors can be created with the **vector()** function.

Numbers

Numbers in R a generally treated as numeric objects (i.e. double precision real numbers)

SOS: If you explicitly want an integer, you need to specify the **L suffix**

e.g.: Entering **1** gives you a **numeric** object;

entering **1L** explicitly gives you an **integer**.

There is also a special number **Inf** which represents **infinity**; e.g. **1 / 0;**

Inf can be used in ordinary calculations; e.g. **1 / Inf is 0**

The value **NaN** represents an undefined value ("**not a number**"); e.g. **0 / 0;**

NaN can also be thought of as a **missing value**

Attributes

R objects can have attributes
- Names
- Dimensions (e.g. matrices, arrays)
- Class
- Length
- Other user-defined attributes and metadata

Attributes of an object can be accessed using the **attributes()** function.

Entering Input

At the R prompt we type expressions. The **“<-”** symbol is the **assignment operator.**

```
x <- 5
> print(x)
[1] 5
>x
[1] 5
> msg <- "hello"
> class(msg)
[1] "character”
```

The **“#”** character indicates a **comment**.
Anything to the right of the # (including the # itself) is ignored.

```
#THIS IS A COMMENT
#######THIS IS A COMMENT
```

Evaluation

When a complete expression is entered at the prompt,
it is evaluated and the result of the evaluated expression is returned.

The result may be auto-printed.

```
> x <- 5        ## nothing printed
> x             ## auto-printing occurs
[1] 5
> print(x)      ## explicit printing
[1] 5
```

The [1] indicates that x is a vector and 5 is the first element.

Printing

```
x <- 1:20
>x
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

The ":" operator is used to create integer sequences.

Vectors

The **c() function** can be used to create vectors of objects.

```
> x <- c(0.1, 0.3)          ##numeric
> x <- c(TRUE, FALSE)       ##logical
> x <- c(T, F, T, T)        ##logical
> x <- c("a", "b", "e")     ## character
> x <- 5:15                 ## integer
>x <- c(1+0.2i, 2+4i)       ## complex
```

Using the vector() function

```
> x <- vector("numeric", length = 10)
>x
[1] 0 0 0 0 0 0 0 0 0 0
```

>>Quiz: can you create a vector of class “integer” using the first expression?

Mixing Objects #DON’T

```
> x <- c(3.5, ”ca")         ## character
> x <- c(F, 2)              ## numeric
> x <- c("a", FALSE)        ## character
```

SOS: When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.
Remember that the only rule about vectors says this is not allowed.
.

Explicit Coercion

If needed, objects can be coerced from one class to another using the **as.*()** functions e.g.

```
> x <- 0:7
print(x)
[1] 0 1 2 3 4 5 6 7
> class(x)
[1] "integer"

> as.numeric(x)
[1] 0 1 2 3 4 5 6 7

> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6" "7"
```

Nonsensical coercion results in NAs.

```
x <- c(”d", ”e", ”f")
> as.numeric(x)
[1] NA NA NA
Warning message: NAs introduced by coercion
> as.logical(x)
[1] NA NA NA
```

Matrices

Matrices are vectors with a dimension attribute.
The dimension attribute is an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,] NA NA NA
[2,] NA NA NA

> dim(m)
[1] 2 3

> attributes(m)
$dim [1] 2 3
```

Matrices

Matrices are constructed column-wise,
so entries start at the “upper left” corner and running down the columns across the “table”.

```
> m <- matrix(3:8, nrow = 2, ncol = 3)
> m
    [,1] [,2] [,3]
[1,]   3    5    7
[2,]   4    6    8
```

cbind and *rbind*

column-binding or row-binding with **cbind()** and **rbind()** respectively can result in matrices.

```
> x <- 3:5
> y <- 11:13

> cbind(x, y)
      x   y
[1,] 3  11
[2,] 4  12
[3,] 5  13


> rbind(x, y)
    [,1] [,2] [,3]
X    3    4    5
y   11   12   13
```

Lists

Lists: special type of vector that can contain elements of different classes.

Can be created with **list()** function

```
> x <- list(1, ”c", FALSE, 3 + 2i)
>x
[[1]]
[1] 1

[[2]]
[1] ”c"

[[3]]
[1] FALSE

[[4]]
[1] 3+2i
```

Inspection:

```
 length(x)
[1] 4

> x[[1]]
[1] 1

> x[[3]]
[1] FALSE

> class(x)
[1] "list”

> class(x[[1]])
[1] "numeric"
```

Factors

Factors are used for **categorical data.**

They can be **ordered or unordered.**

Factors are very useful in modelling functions like **lm()** and **glm()**

Using factors with labels is preferred than using integers, since factors are self-describing;

→Having a variable that has values “Patient” and “Control” is better
than having a variable that has values 1 and 2.

Factors

Can be created using the **factor()** function

```
 x <- factor(c("female", "female", "male", "female", "male"))
> x
[1] female female male   female male
Levels: female male

> table(x)
x
female        male
     3           2
```

Missing Values

Missing values NA or NaN for undefined mathematical operations.

is.na() function can be used to test objects for NA values
is.nan() function can be used to test objects for NaN values

```
> x <- c(5, 10, NA, 120, 70)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE

> x <- c(5, 10, NaN, NA, 40)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

Data Frames

Data frames contain tabular data.
Essentially, they are a special type of **list** where every element (“column”) of the list has the same length
We can store different classes of objects in each column of a data frames;
(but all the elements of a matrix has to be of the same class)
read.table() or **read.csv()** functions can be used to create data frames
They can be converted to a matrix with **data.matrix()** function

```
> x <- data.frame(tik = 5:8, tok = c(T, T, F, F), boom=c("no", "time", "to", "die" ))
> x
   tik    tok      boom
1   5     TRUE     no
2   6     TRUE     time
3   7     FALSE    to
4   8     FALSE    die

> nrow(x)
[1] 4

> ncol(x)
[1] 3
```

Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 8:10
>attributes(x)
NULL

> names(x)
NULL

> names(x) <- c("tik", "tok", "boom")
>x
tik  tok  boom
8    9    10

> names(x)
[1] " tik" "tok" "boom"
```

Subsetting

Operations used to subtract, isolate, extract specific portions (subsets) of R objects

Subsetting with “ […] “ returns an object of the same class as the original object by default

```
## Subsetting a vector
> x <- c ("no", "time", "to", "die")
> x[2]
[1] ”time"
> x [2:4]
[1] "time" "to" "die”
```

```
## Subsetting a Matrix
> x <- matrix(5:10, nrow=3, ncol=2)
> x[2, 1]
[1] 8
> x[3, 2]
[1] 10
>x[3,1]
[1] ??
```

```
## Subsetting with missing indices
> x[2,  ]
[1] 6  9
> x[  , 2]
[1] 8  9  10

##return the whole 3rd row
>x[ ? , ? ]
```

Subsetting

Subsetted single elements from a matrix are returned as vectors (length =1)
Subsetted single columns or single rows will return a vector, not a matrix.
Setting drop = FALSE turns off this setting

```
> x <- matrix(5:10, 3, 2)  ##Note: nrow= and ncol= can be omitted
```

```
> y<-x[1, 2]
> y
[1] 8
> class(y)
[1] "integer”
```

```
> y<-x[  , 2]
> y
[1] 8  9  10
> class(y)
[1] "integer”
```

```
> y<- x[1, 2, drop = FALSE]
> y
       [   ,1]
[1,   ]    8
>class (y)
[1] "matrix"
```

```
> y<- x[  , 2, drop= F]
>y
       [ ,1 ]
[ 1, ]     8
[ 2, ]     9
[ 3, ]     10
>class (y)
[1] "matrix"
```

Subsetting

Subsetting Lists and Dataframes
(can be tricky!)

Subsetting with " **[[…]]** " extracts elements from a **list** or a **dataframe**. The class of the returned object can be different compared to the source element

Subsetting with " **$** " extracts elements from a **list** or a **dataframe BY NAME** (handy).

```
x <- list(no= FALSE , time = 5:10, die = 100.01)
```

```
> x[2]
$time
[1]  5  6  7  8  9 10

> x[[2]]
[1 ]  5  6  7  8  9 10
```

```
> x$die
[1] 100.01

> x[["die"]]
[1] 100.01

> x["die"]
$die
[1] 100.01
```

```
##Extracting multiple elements of a list:
> x[c(1,2)]

$no
[1] FALSE

$time
[1]  5  6  7  8  9 10
```

Subsetting

" **[[ ]]** " can be used with computed indices, e.g. variables from the environment
" **$** " can **only** be used with names

x <- list(**no= FALSE** , time = 5:10, die = 100.01)

```
> new_name <- "no"

> x$new_name                ## new_name does not exist in list x
NULL

> x[[new_name]]             ## computed index for 'new_name'
[1] FALSE

> x$time                    ## 'time' does exist in list x
[1]  5  6  7  8  9 10
```

## Subsetting

##Subsetting Nested Elements

```
> x <- list(a = c(5,10,15), b = c(50.5, 100.01))
> x[[c(1,3)]]
[1] 15

> x[[1]][[3]]
[1] 15
```

## " **[[ ]]**" and " **$** " operators also allow partial matching of names

```
> x <- list(no= FALSE , time = 5:10, die = 100.01)

> x$ti
[1]  5  6  7  8  9 10

> x[["ti"]]
NULL

> x[[tim]]
Error: object 'tim' not found

> x[["tim", exact = FALSE]]
[1]  5  6  7  8  9 10
```

Remove NAs

Missing values (NAs) are commonly removed

Using the **is.na()** function

```
> x <- c(5, 6, 7, NA,  9, NA)
> logic_vect_y<- is.na(x)
> logic_vect_y
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE
> y<-x[! logic_vect_y]
```

" **!** " operator is translated as "the opposite of"
## **logic_vect_y** here is a computed index (variable from the environment)

```
> y
[1] 5 6 7 9
```

Using the **complete.cases()** function

```
> x <- c(5, 6, 7, NA,  9, NA)
> logic_vect_y<- complete.cases(x)
> logic_vect_y
[1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE
> y<-??
```

Using the **na.omit()** function

```
>x <- data.frame(x1 = c(9, 6, NA, 9, 2, 5, NA),
                 x2 = c(NA, 5, 2, 1, 5, 8, 0),
                 x3 = c(1, 3, 5, 7, 9, 7, 5))
>data_NA_omit<- na.omit(x)
  x1 x2 x3
2  6  5  3
4  9  1  7
5  2  5  9
6  5  8  7
```

Vectorized Operations

```
>  a<-7:10;  b<-1:4
>  a+b
[1]  8 10 12 14

>  a > 8
[1] FALSE FALSE  TRUE  TRUE

>  b >= 2
[1] FALSE  TRUE  TRUE  TRUE

> a * b
[1]  7 16 27 40

> a / b
[1] 7.0 4.0 3.0 2.5
```

```
> m1<- matrix(5:10, 3, 2) ; m2<- matrix(rep(10,6), 3,2)
> m1
      [,1] [,2]
[1,]    5    8
[2,]    6    9
[3,]    7   10
> m2
      [,1]  [,2]
[1,]   10   10
[2,]   10   10
[3,]   10   10

> m1 * m2    ##element wise multiplication
      [,1]  [,2]
[1,]   50   80
[2,]   60   90
[3,]   70  100
```

Importing data

read.table(), for reading tabular data, usually from txt (tab separated)

read.csv(), for reading tabular data , usually from spreadsheets (comma separated)

readLines(), for reading lines of a text file

source(), for reading in R code files - inverse of dump()

dget(), for reading in R code files - inverse of dput()

load(), for reading in saved workspaces – inverse of save()

Writing files

write.table(), for writing tabular data

writeLines(), for writing lines to a text file

dump(), for writing R code files - inverse of source()

dput(), for reading in R code files - inverse of dget()

save(), for saving workspaces – inverse of load()

read.table()

One of the most commonly used functions in R

Useful arguments:

file, the name of a file, or a connection

header, (logical) is there a header line in the file?

sep, ("string") how the columns are separated? e.g. "\t", " " (space) etc

colClasses, (character vector) the class of each column in the dataset

nrows, (integer) the number of rows in the dataset

skip, (integer) lines to skip from the beginning of the file

stringsAsFactors, (logical) should character variables treated as factors?

####**read.csv**() is identical to read.table except that the default separator is a comma

Let's recap

Set the working directory (where sample1_data.csv is stored)
Read in the sample1_data.csv file and store it as "input"

1) What are the column names of the dataset?
2) How many rows and how many columns are there?
3) Store the first 5 rows of the dataframe in a new object named "top5". Print top5 to the console
4) Store the last 8 rows of the dataframe in a new object named "last8". Print last8 to the console
5) What is the value of the large particles at the 56th row?
6) What is the average value of microparticles after removing the NAs?
7) What is the average temperature during the days with humidity < 75% ?
8) Reset the row names of the subset df
9) Return the month and day when we observe the maximun value of microparticles ?
    (which.max() might be useful)

Answers

```
> setwd("/…/…/…/")
> input<- read.csv("sample1_data.csv", header=T, stringsAsFactors=FALSE)
> colnames(input)
> ncol(input)
> nrow(input)
> top5 <- head(input, 5)      or      >top5 <- input[1:5,]
> last8 <- tail(input, 8)  ##elegant   or    > input[(nrow(input)-8):nrow(input),] ## not so elegant
> input[56,"large_particles"]   or   > input[56,4]
> mean(na.omit(input$microparticles))   ##(37.60656)
> subset_hum_less75<-input[input$humidity<75,]
> mean(subset_hum_less75$temperature)
Or in one step
>mean(input$temperature[input$humidity<75])
> row.names(subset_hum_less75) <- NULL
> max(na.omit(input$microparticles))       or       > max(input$microparticles, na.rm = TRUE)
> subset_max_temp<-input[input$temperature == max(input$temperature),]
> subset_max_temp$Month
> subset_max_temp$Day
```

Control Structures

if, else: testing a condition

for: execute a loop a given number of times

while: execute a loop while a condition is true

repeat: execute an infinite loop

break: break the loop when a condition is true

next: skip an iteration of a loop

return: return the result of a function

if, else, else if

```
## Two cases control
if(…condition to meet…) {
## do something here

} else {
## do something else here

}
```

```
##Example:

if(x > 5) {
y <- 8

} else {
y <- 10

}
```

```
## Three cases control
if (…condition to meet…) {
## do something

} else if (…another condition to meet…) {
## do something different

} else {
## do something different }
```

for

for loops can be used for iterating over the elements of a vector, a column of a dataframe etc. They take an iterator and use the incrementing value as an "index"

```
for(i in 1:50) {
print(i+5)
}


x <- c("no", "time", "to ", "die")

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

Nested *for* loops

```
x <- matrix(1:9, 3, 3)
for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

while

While loops iterate while a condition remains true. Testing is done before each iteration

```
startingVariable <- 0 ##initialise the variable
while(startingVariable < 50) {
print(startingVariable)
startingVariable <- startingVariable + 2
}

z <- 10
while(z >= 5 && z <= 15) {
 print(z)
 coin <- rbinom(1, 1, 0.5)
        if(coin == 1) {  ## random walk
         z <- z + 1
        } else {
         z <- z - 1
 } }
```

R - Programming in Biosciences

Dr. Timokratis Karamitros
Researcher

Head, Unit of Bioinformatics and Applied Genomics
Hellenic Pasteur Institute, Athens, GR

tkaram@pasteur.gr

FUNCTIONS

They are created using
function()
They are stored as R objects (of class “function”)

```
f <- function(arguments) {
      ###execute computations based on the arguments
}
```

Functions can be arguments or can be defined inside of another function

Arguments

Functions have named arguments which may have default values, or not

The *formal arguments* are the arguments defined in the function
The *formals()* function returns the formal arguments of a function
Some arguments can be missing
Arguments can be matched by position or by name

```
> range1 <- c(10:50)

> sd(range1)
[1] 11.97915
> sd(x = range1)
[1] 11.97915
> sd(x = range1, na.rm = FALSE)
[1] 11.97915
> sd(na.rm = FALSE, x = range1)
[1] 11.97915
> sd(na.rm = FALSE, range1)
[1] 11.97915
```

Arguments

```
> args(glm)
function (formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart, offset, control = list(...),
    model = TRUE, method = "glm.fit", x = FALSE, y = TRUE, singular.ok = TRUE,
    contrasts = NULL, ...)
NULL
```

Try > formals(glm)

Function definition and calling

```
f <- function(x, y = 5, z = 2) {
(x^z)+y
}

>f(5)
[1] 30

>f(5,y=8)
[1] 33
```

The “...” Argument

The "..." argument defines a variable number of arguments that can be passed to other functions.

Can be used to "modify" other functions while preserving a long list of arguments unchanged

```
##examine read.table() arguments
>args(read.table)


>f<-function(file, sep="," , ... ) {

read.table(file, sep=sep, ... )
}



##Compare
>read.table("sample1_data.csv")
##with
>f("sample1_data.csv")
```

The “...” Argument

The "..." argument ca be used to define an unknown number of arguments

```
> args(paste)

function (..., sep = " ", collapse = NULL) ###use collapse in paste0()

> paste("no", "time", "to", "die", sep = "-")
[1] "no-time-to-die"
```

SOS: Arguments that appear after "..." must be named and cannot be partially matched or positionally defined

Loop functions

R hosts a functions family to perform loops in a "vectorised" way --> less code writing (more to remember)

lapply() / sapply() : Apply a function over a **L**ist (on each element)

apply(): Apply a function over the margins of an **A**rray, usually a m**A**trix

tapply(): Apply a function over subsets of a vec**T**or

mapply(): **M**ultivariate version of lapply

lapply() / sapply()

```
> lapply (X, FUN, ...)

X :       a list. If not, it will be coerced to a list
FUN :     a function
... :     other arguments
```

##**l**apply always returns a **L**ist

```
> x <- list(a = 5:10, b = rnorm(5), c = rnorm(5, 2), d = rnorm(4, 1, 1))
> lapply(x, mean)
$a
[1] 7.5
$b
[1] 0.100552
$c
[1] 1.578663
$d
[1] 1.202836
```

## **s**apply() returns a **S**implified result, if possible

```
> sapply(x, mean)
        a         b         c         d
7.5000000 0.1005521 1.5786633 1.2028363
```

apply()

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

X :	an array
MARGIN :	a vector indicating which margins to be analysed
FUN :	a function to be applied
...	other FUN arguments

```
> m <- matrix(1:200, 20, 10)

> apply(m, 1, mean)
 [1]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110

> apply(m, 2, mean)
 [1]  10.5  30.5  50.5  70.5  90.5 110.5 130.5 150.5 170.5 190.5
```

```
##ready to use shortcuts (faster):
rowSums = apply(x, 1, sum)
rowMeans = apply(x, 1, mean)
colSums = apply(x, 2, sum)
colMeans = apply(x, 2, mean)
```

tapply()

tapply() is used to apply a function over subseted vectors

```
> tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

X :	a vector
INDEX :	a factor or a list of factors
FUN :	a function
... :	other FUN arguments
simplify :	logical, simplification of the result

```
> v <- c(rnorm(5), rnorm(5, 2), rnorm(5, 5))
> f <- gl(3, 5)    ## Generate factors by specifying the pattern of their levels.
> f
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3


> tapply(v, f, mean)
        1         2         3
0.183946  1.593078  4.650117
```

lapply() with split()

```
##split()  splits e.g. a vector or a dataframe into groups based on a factor

> v <- c(rnorm(5), rnorm(5, 2), rnorm(5, 5))
> f <- gl(3, 5)
> v
 [1] -1.0318472  0.2218044  0.4246628  0.4485537  0.8565563
 [6]  1.7576478  2.3410273  1.6256701  0.4911859  1.7498583
[11]  2.8567678  5.4816134  6.1299922  3.7236435  5.0585677

> split(v, f)
$`1`
[1]          -1.0318472  0.2218044   0.4246628   0.4485537   0.8565563

$`2`
[1]          1.7576478   2.3410273   1.6256701   0.4911859   1.7498583

$`3`
[1]           2.856768    5.481613    6.129992    3.723644    5.058568
```

```
##(previously)
> tapply(v, f, mean)
         1          2          3
0.183946   1.593078   4.650117

##lapply() with split()
> lapply(split(v, f), mean)
$`1`
[1] 0.183946

$`2`
[1] 1.593078

$`3`
[1] 4.650117
```

lapply() with split() on a dataframe

```
> dataframe1<-PlantGrowth   ###(R built-in data)
  weight group
1   4.17  ctrl
2   5.58  ctrl
3   5.18  ctrl
----
11   4.81  trt1
12   4.17  trt1
13   4.41  trt1
----
21   6.31  trt2
22   5.12  trt2
----
```

```
###split the dataframe based on $group
S<-split(dataframe1, dataframe1$group)

> S
$ctrl
  weight group
1   4.17  ctrl
2   5.58  ctrl
----
10   5.14  ctrl

$trt1
  weight group
11   4.81  trt1
----
19   4.32  trt1
20   4.69  trt1

$trt2
  weight group
21   6.31  trt2
----
30   5.26  trt2
```

```
>S<-split(dataframe1, dataframe1$group)
> lapply(S, function(S) mean(S[ , "weight"]))
$ctrl
[1] 5.032

$trt1
[1] 4.661

$trt2
[1] 5.526
```

R- Programming in Biosciences

Dr. Timokratis Karamitros
Researcher

Head, Unit of Bioinformatics and Applied Genomics
Hellenic Pasteur Institute, Athens, GR

tkaram@pasteur.gr

Plotting in R

Plotting

base and recommended packages:

graphics: plotting functions for the "base" graphing systems, including *plot, hist, boxplot*
each aspect of the plot handled separately through a series of function calls; this is often conceptually simpler

lattice: independent of the "base" graphics system; includes functions like *xyplot, histogram, etc*
usually created in a single function call, so all of the graphics parameters have to specified at once

grDevices: various graphics devices, including X11 (Unix), windows (windows) quartz (Mac), PDF, PostScript, PNG, etc.
usually not directly controlled by the end user

ggplot: ggplot() initializes a ggplot object. It can be used to declare the input data frame for a graphic and
to specify the set of plot aesthetics intended to be common throughout all subsequent layers
unless specifically overridden.

Base Graphics

Important Base Graphics Parameters

pch: the plotting symbol (default is open circle)
lty: the line type (default is solid line), can be dashed, dotted, etc.
lwd: the line width, specified as an integer multiple
col: the plotting color, specified as a number, string, or hex code
las: the orientation of the axis labels on the plot
bg: the background color
mar: the margin size
oma: the outer margin size (default is 0 for all sides)
mfrow: number of plots per row, column (plots are filled row-wise)
mfcol: number of plots per row, column (plots are filled column-wise)

Base Graphics

Important Base Graphics Functions

plot(): make a scatterplot, or other type of plot
lines(): add lines to a plot, given a vector x values and a corresponding vector of y values
points(): add points to a plot
title: add annotations to x, y axis labels, title, subtitle, outer margin
mtext: add arbitrary text to the margins (inner or outer) of the plot axis: adding axis ticks/labels

Base Graphics

Some Examples

```
x<-rnorm(50)
y<-rnorm(50)
hist(y)
plot(x,y)

z<-rnorm(50,1,2)

plot(x,z)
example(points)
plot(x,z, pch = 17)

plot(x,y, main="testScatter") ##again
fitmodel<-lm(y~x)
abline(fitmodel,col="red")
```

Base Graphics

Plotting in panels

```
par(mfcol=c(2,2))
#or
par(mfrow=c(2,2))


plot(x, y, main="testScatter 1", pch=0)
fitmodel1<-lm(y~x)
abline(fitmodel1,col="red")

plot(x, z, main="testScatter 2", pch=1)
fitmodel2<-lm(z~x)
abline(fitmodel2,col="blue")

plot(z, y, main="testScatter 3", pch =2)
fitmodel3<-lm(y~z)
abline(fitmodel3,col="green")

plot(y, x, main="testScatter 4", pch =3)
fitmodel4<-lm(x~y)
abline(fitmodel4,col="orange")
```

Lattice Graphics

```
library(lattice)
##load some data
data (environmental)
head (environmental)


xyplot(ozone ~ radiation, data=environmental)
xyplot(ozone ~ temperature, data=environmental)

##divide the temperature vector into 6 parts containing equal number of partially overlapping observations
temp.zones<-equal.count(environmental$temperature,6)
temp.zones

xyplot(ozone ~ radiation | temp.zones, data=environmental, layout=c(1,6), pch=17)
splom(~environmental)
wind.zones<-equal.count(environmental$wind,4)
histogram(~ozone | temp.zones * wind.zones, data=environmental)
```

ggplot Graphics

ggplot2 is a member of the "tidyverse" collection of R packages

```
>install.packages("tidyverse")   ##or just
>install.packages("ggplot2")
```

More to read on tidyverse:
https://r4ds.had.co.nz

ggplot() initializes a ggplot object.

It can be used to declare the input data frame for a graphic and to specify the set of plot aesthetics intended to be common throughout all subsequent layers unless specifically overridden.

A statistical graphic is a **mapping** from data to **aesthetic** attributes (colour, shape, size) of **geometric** objects (points, lines, bars).
ggplot splits the difference between *base* and *lattice*

- Automatically deals with spacings, text, titles but also allows you to annotate by “adding”
- Superficial similarity to lattice but generally easier/more intuitive to use

http://vita.had.co.nz/papers/layered-grammar.pdf

ggplot Graphics

qplot()

Looks for data in a data frame, similar to lattice
Plots are made up of *aesthetics* (size, shape, color) and *geoms* (points, lines)
Factors are important for indicating subsets of the data, they should be **labelled**

```
> library(ggplot2)
> str(mpg)
> head(mpg)
> qplot(displ,hwy, data=mpg)
> qplot(displ,hwy, data=mpg, geom = c("point", "smooth"))
> qplot(displ,hwy, data=mpg, color=class)
> qplot(displ, data = mpg, fill = class)
> qplot(displ, hwy, data = mpg, facets = . ~ drv)
> qplot(displ, data = mpg, fill = class)
```

ggplot Graphics

ggplot() is the core function and offers more flexibility compared to qplot()

```
ggplot(data = mpg) +
        geom_point(mapping = aes(x = displ, y = hwy))
```

ggplot() creates a coordinate system that you can add layers to
geom_point() adds a layer of points to your plot, which creates a scatterplot
The mapping argument defines how variables in your dataset are mapped to visual properties.
The mapping argument is always paired with aes()
The x and y arguments of aes() specify which variables to map to the x- and y-axes

ggplot Graphics

qplot()

Looks for data in a data frame, similar to lattice
Plots are made up of *aesthetics* (size, shape, color) and *geoms* (points, lines)
Factors are important for indicating subsets of the data, they should be **labelled**

ggplot() is the core function and offers more flexibility compared to qplot()

```
> library(ggplot2)
> str(mpg)
> head(mpg)
> qplot(displ,hwy, data=mpg)
```

ggplot Graphics

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class, size = cyl))

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class, alpha = cyl))

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class, shape = class))

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, ncol = 4)

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class)) +
  geom_abline()+
  facet_grid(drv ~ cyl)
```

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = class)) +
  geom_boxplot()
```

R - Programming in Biosciences

Dr. Timokratis Karamitros
Researcher

Head, Unit of Bioinformatics and Applied Genomics
Hellenic Pasteur Institute, Athens, GR

tkaram@pasteur.gr

100 YEARS | HELLENIC PASTEUR INSTITUTE