Introduction to R

Alberto Garfagnini

Università di Padova

R lecture 1

What is R ?

- R is a language and environment for statistical computing and graphics
- similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. Evolution of S, R open source
- R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, ...) and graphical techniques
- R is highly extensible how create package (of function eg.)
- R is available as Free Software (GNU GPL) and it compiles and runs on a wide variety of UNIX platforms, Windows and MacOS
- The latest R version is 3.6.3 (Holding the Windsock), released on February 29th, 2020 (20 years after R 1.0.0 release)

R Web Resources

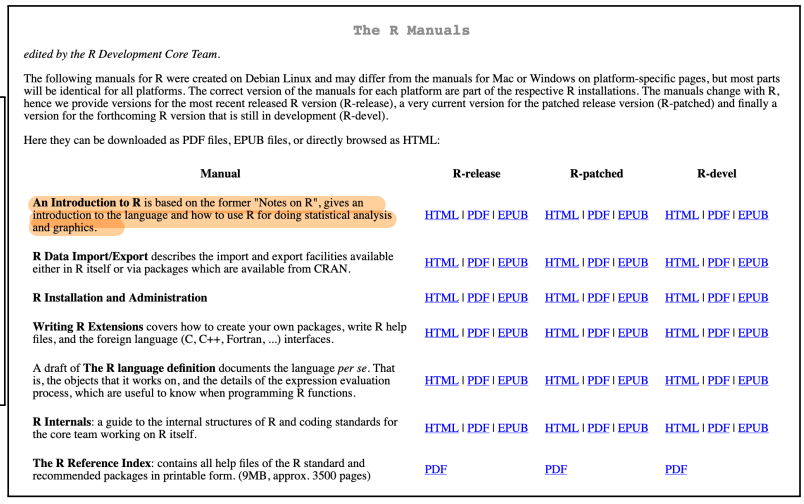
- R Web Site: https://www.r-project.org/
- R source code: https://cran.r-project.org/src/base/R-3/
- A list of changes in the new version can be found here: https://cran.r-project.org/doc/manuals/r-release/NEWS.html

R is 20 years old

- the first version of R (1.0.0) was released on Feb 29, 2000
  - original release message: https://stat.ethz.ch/pipermail/r-announce/2000/000127.html
- the 20 years celebration took place in Copenhagen, (28-29 Feb 2020)
- http://www.celebration2020.org/
- https://www.youtube.com/channel/UCqEdfW-1KUn_QQyQogxqLeA/ Presentation w ft history

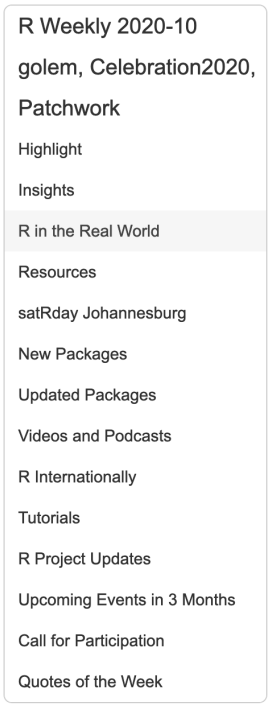
CRAN: the Comprehensive R Archive Network

- Access point to R resources: HOWTOs, FAQ, manuals, examples, ...
- CRAN Web Page: https://cran.r-project.org/
- a list of Frequently Asked Questions is available https://cran.r-project.org/faqs.html
- an open access R journal is published online once/twice per year: https://journal.r-project.org/
- and several Manuals are available on CRAN Web Page:

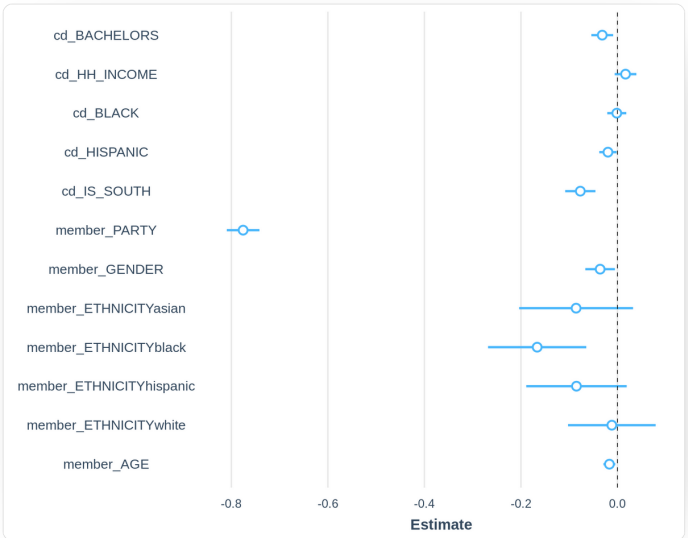


RWeekly.org

- It's a Web repository with lots of great blogs, tutorials and other formats of resources coming out every day



- A SHINY QUIZ APP ABOUT THE RUSSIAN INFLUENCE CAMPAIGN BEFORE THE 2016 US ELECTIONS (skranz.github.io)
- Predicting the video game hype train - Playing around with Naïve Bayesian Learning (rcrastinate.rbind.io)
- COVID-19 epidemiology with R (rviews.rstudio.com)
- When does "garbage time" in an start in an NBA game? (jtcies.com)
- Modeling roll call voting behavior in the US House (jtimm.net)



How to install R

Local Installation

- from sources: `https://cran.stat.unipd.it/src/base/R-3/R-3.6.3.tar.gz`
- or using pre-defined packages for
  - Linux (check with your favorite distribution)
  - mac OS X (for Catalina and Legacy Os Systems)
  - Windows, (`https://cran.stat.unipd.it/bin/windows/base/`)

Anaconda distribution

- a free and open-source distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment
  - it uses `Conda`, an open source, cross-platform, language-agnostic package manager and environment management system.
  - it is available for Linux, macOS and Windows: `https://www.anaconda.com`

Using Virtualization tools

- with Docker (`https://www.docker.com`), using predefined containers
  - `docker pull r-base` for R 3.6.3, alone
  - `docker pull Jupiter/r-notebook`, for R 3.6.1 integrated with Jupyter

R console with Docker

- List images:

```
$ docker images -a
REPOSITORY          TAG       IMAGE ID       CREATED       SIZE
...
r-base              latest    3aad1ffccc53   7 days ago    679MB
```

- check computer IP and enable remote host display

```
$ open -a xquartz
$ ifconfig | grep 192
        inet 192.168.38.204 netmask 0xffffff00 broadcast 192.168.38.255
$ xhost + 192.168.38.204
192.168.38.204 being added to access control list

$ docker run -it --rm -v "$PWD":/mnt \
         -e DISPLAY=192.168.38.204:0 --name rintera r-base

R version 3.6.3 (2020-02-29) -- "Holding the Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
...
> plot(1:10,1:10, col="pink")
> q()
Save workspace image? [y/n/c]: n
$
```


Jupyter notebook with Docker

- List images:

```
$ docker images -a
REPOSITORY          TAG       IMAGE ID       CREATED       SIZE
jupyter/r-notebook  latest    eca5843b30ea   2 days ago    3.37GB
...
```

- Start Docker container in 'detached' mode

```
$ docker run -d -P --rm --name nb_R01 \
         -v "$PWD":/home/jovyan/work jupyter/r-notebook
```

- Check running container. Extract HTTP port

```
$ docker ps -a
CONTAINER ID  IMAGE                [..] PORTS                     NAMES
0a36c87d6cd6  jupyter/r-notebook   [..] 0.0.0.0:32770->8888/tcp   nb_R01
```

- Inspect the container log file → extract the Jupyter token for Web login

```
$ docker logs --tail 3 nb_R01
    Or copy and paste one of these URLs:
        http://0a36c87d6cd6:8888/?token=94ed...8b52
     or http://127.0.0.1:8888/?token=94ed...8b52
```

- Open the page in Browser. Once asked, insert the Jupyter token

```
http://127.0.0.1:32770
```

How to run R ?

Two running modes are available:

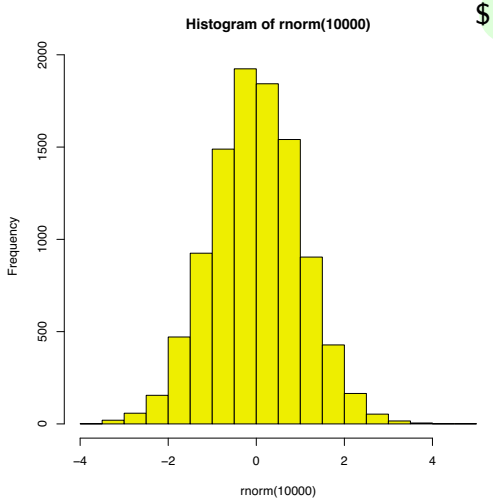
- interactive mode
- batch mode

Interactive mode R

```
$ R
> pdf("xh.pdf")
> hist(rnorm(1000),
       col="yellow")
> dev.off()
null device
          1
```

Save pdf

pdf(" ... ")
hist()
dev.off()



Batch mode R

Run file

```
file: xh_plot.R
pdf("xh.pdf")
hist(rnorm(1000), col="yellow")
dev.off()
```

```
$ R CMD BATCH xh_plot.R
```

Execute the file

← file: xh.pdf

- test.Rout → time to run it
  results

Starting an interactive R session

- the R program can be invoked from the bash shell

```
$ R

R version 3.6.3 (2020-02-29) -- "Holding␣the␣Windsock"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (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.

>
```

. . . and ending it

- exiting R can be done through the `q()` function or by typing `<CTRL>-d`

```
> q()
Save workspace image? [y/n/c]:
```

- at the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started

Getting help in R

- the simplest way, if the name of the function we need help with is known, it to prefix it with the question mark symbol (?)

```
> ?plot

plot                package:graphics                R Documentation

Generic X-Y Plotting

Description:

  Generic function for plotting of R objects.  For more details
  about the graphical parameter arguments, see 'par'.

  For simple scatter plots, 'plot.default' will be used.  However,
  there are 'plot' methods for many R objects, including
  'function's, 'data.frame's, 'density' objects, etc.  Use
  'methods(plot)' and the documentation for these.

Usage:

     plot(x, y, ...)
...
```

Getting help in R

help(lust)

- if the name of the function is not known, but only the subject on which help is needed, the `help.search()` function can be used

```
> help.search("data input")

Help files with alias or concept or title matching 'data input'
using fuzzy matching:


utils::read.DIF          Data Input from Spreadsheet
utils::read.table        Data Input


Type '?PKG::FOO' to inspect entries 'PKG::FOO', or 'TYPE?PKG::FOO'
for entries like 'PKG::FOO-TYPE'.
```

- or with the `find()` and `apropos()` functions

```
> find("read.table")
[1] "package:utils"

> apropos("lm")
 [1] ".colMeans"      ".lm.fit"        "KalmanForecast" "KalmanLike"
 [5] "KalmanRun"      "KalmanSmooth"   "colMeans"       "confint.lm"
 [9] "contr.helmert"  "dummy.coef.lm"  "getAllMethods"  "glm"
[13] "glm.control"    "glm.fit"        "kappa.lm"       "lm"
[17] "lm.fit"         "lm.influence"   "lm.wfit"        "model.matrix.lm"
[21] "nlm"            "nlminb"         "predict.glm"    "predict.lm"
[25] "residuals.glm"  "residuals.lm"   "summary.glm"    "summary.lm"
```


R worked examples

- all R functions have a set of working examples that can be invoked and examined

```
> example(sqrt)

sqrt> require(stats) # for spline

sqrt> require(graphics)

sqrt> xx <- -9:9

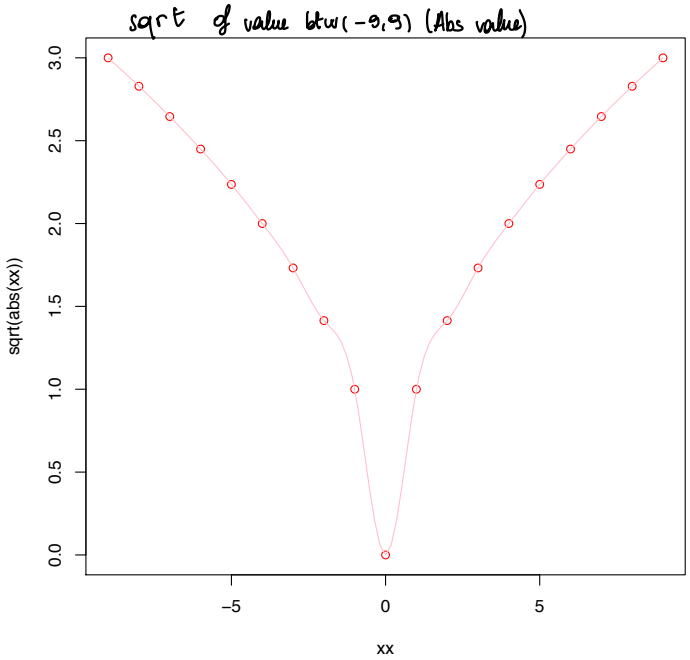
sqrt> plot(xx, sqrt(abs(xx)),
           col = "red")

Hit <Return> to see next plot:

sqrt> lines(spline(xx,
                   sqrt(abs(xx)),
                   n=101),
            col = "pink")
```

Will show example

interpolation



R packages - help

- functions and sets of data are organized in packages
- to find help and list the contents of a packages, the `library(help=package.name)` function will give details on the packages and a list of all the functions and data sets.

```
> library(help=base)
```

call a package

```
                Information on package 'base'
Description:

Package:        base
Version:        3.6.3
Priority:       base
Title:          The R Base Package
Author:         R Core Team and contributors worldwide
Maintainer:     R Core Team <R-core@r-project.org>
Description:    Base R functions.
License:        Part of R 3.6.3
Suggests:       methods
Built:          R 3.6.3; ; 2020-02-29 10:11:03 UTC; unix

Index:

.Call                   Modern Interfaces to C/C++ code
...
zapsmall                Rounding of Numbers
```


R packages - listing

tibble library(tibble)

- with the command `installed.packages()` it is possible to retrieve a list of all the installed packages

Can save the result inside a variable
a ← installed.packages()

```
> pkg <- installed.packages()
> df_pkg <- data.frame(pkg)
> names(df_pkg)
```

create dataframe

```
 [1] "Package"           "LibPath"           "Version"
 [4] "Priority"          "Depends"           "Imports"
 [7] "LinkingTo"         "Suggests"          "Enhances"
[10] "License"           "License_is_FOSS"   "License_restricts_use"
[13] "OS_type"           "MD5sum"            "NeedsCompilation"
[16] "Built"
```

```
> length(df_pkg[,1])
[1] 31
```

```
> df_pkg[c(1:9,25:31),c(1,3,5,10)]
            Package Version                        Depends            License
docopt       docopt   0.6.1                           <NA> MIT + file LICENSE
littler     littler   0.3.9                           <NA>         GPL (>= 2)
base           base   3.6.3                           <NA>    Part of R 3.6.3
boot           boot  1.3-24 R (>= 3.0.0), graphics, stats          Unlimited
class         class  7.3-15    R (>= 3.0.0), stats, utils      GPL-2 | GPL-3
cluster     cluster   2.1.0                 R (>= 3.3.0)         GPL (>= 2)
codetools codetools  0.2-16                   R (>= 2.1)                GPL
compiler   compiler   3.6.3                           <NA>    Part of R 3.6.3
datasets   datasets   3.6.3                           <NA>    Part of R 3.6.3
splines     splines   3.6.3                           <NA>    Part of R 3.6.3
stats         stats   3.6.3                           <NA>    Part of R 3.6.3
stats4       stats4   3.6.3                           <NA>    Part of R 3.6.3
survival   survival   3.1-8                 R (>= 3.4.0)        LGPL (>= 2)
tcltk         tcltk   3.6.3                           <NA>    Part of R 3.6.3
tools         tools   3.6.3                           <NA>    Part of R 3.6.3
utils         utils   3.6.3                           <NA>    Part of R 3.6.3
```

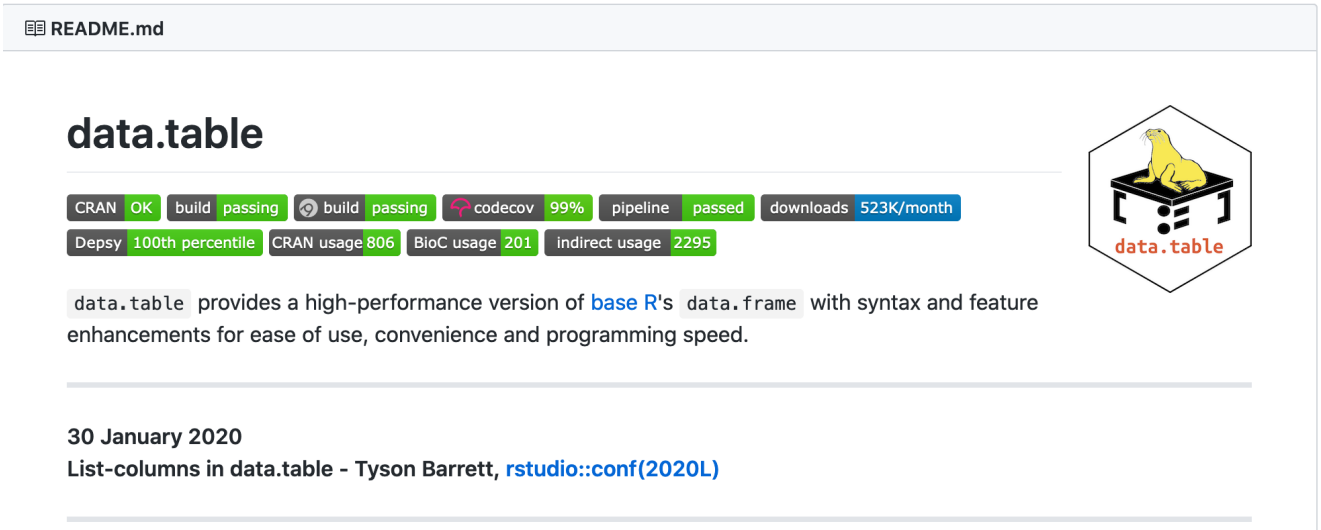
R packages - installing

- a package can be installed from three main sources :

1. from CRAN (official stable versions)
2. from GitHub (developer versions)
3. from other repositories, (for instance BioConductor)

Package: `data.table`

- on CRAN: https://cran.r-project.org/web/packages/data.table/index.html
- on GitHub: https://github.com/Rdatatable/data.table



R session housekeeping

- to list all the objects created with the current session, use the `ls()` or `objects()` functions

You can save an objects

```
> objects()
 [1] "Rdate"    "XXL"      "ctl"      "data"     "dc"       "diffs"
 [7] "dl"       "duration" "group"    "hh"       "lm.D9"    "lm.D90"
[13] "model"    "ncol"     "op"       "opar"     "r"        "res"
[19] "st"       "t1"       "t2"       "test1"    "tf"       "times"
[25] "trt"      "weight"   "x"        "xx"       "y"        "y1"
[31] "y2"
```

- to list all the packages and data frames currently attached to the running R session, use `search()`

```
> search()
 [1] ".GlobalEnv"        "package:lattice"   "times"
 [4] "data"              "package:stats"     "package:graphics"
 [7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"   "Autoloads"         "package:base"
```

R as a calculator

- the screen prompt > invites to type commands and data
- the command line can be used as a calculator

```
> log(34/5.5)
[1] 1.821612
```

- each line can have up to 8192 characters, but can be continued on further lines if incomplete (the prompt will change from > to +)

```
> log(34.7) + sqrt(12) -
+ 25 / 7 * 46^3
[1] -347621.6
```

- two or more expressions can be placed on the same line, if are separated by ';'

```
> log(10); sqrt(3.75)*4.7; 2^2
[1] 2.302585
[1] 9.101511
[1] 4
```


R knows complex numbers

- complex numbers arithmetic's and elementary trigonometric, logarithmic, exponential, square root and hyperbolic functions are implemented
- a complex number has the imaginary part identified by a lower-case 'i'

```
> 3.5 +2i
[1] 3.5+2i
```

- special R functions can be used with complex numbers :

```
> Re(3.5 + 2i)
[1] 3.5
> Im(3.5 + 2i)
[1] 2
> Mod(3.5 + 2i)
[1] 4.031129
> Arg(3.5 + 2i)
[1] 0.5191461
> Conj(3.5 + 2i)
[1] 3.5-2i
> is.complex(3.5 + 2i)
[1] TRUE
> as.complex(3.5)
[1] 3.5+0i
```

Function	Description
Re(z)	Extract the real part
Im(z)	Extract the imaginary part
Mod(z)	Calculate the modulus
Arg(z)	Calculate the argument : Arg(x+yi) = atan(y/x)
Conj(z)	Work out the complex conjugate
is.complex(z)	test for complex number membership
as.complex(z)	force the input as a complex number

R mathematical functions

Function	Description
`log(x)`	base e log of `x`
`exp(x)`	anti-log of `x`
`log(x,n)`	base `n` log of `x`
`log10(x)`	base 10 log of `x`
`sqrt(x)`	square root of `x`
`factorial(x)`	$x! = x(x-1)(x-2)\ldots 3 \cdot 2 \cdot 1$
`choose(n,x)`	binomial coefficient, $n!/(x! \cdot (n-x)!)$
`gamma(x)`	$\Gamma(x)$ for real `x`, $(x-1)!$ for integer `x`
`lgamma(x)`	natural log of $\Gamma(x)$
`abs(x)`	absolute value for `x`
`floor(x)`	greater integer less than `x`
`ceiling(x)`	smallest integer greater than `x`
`trunc(x)`	closest integer to `x` between 0 and `x`; it behaves as `floor()` for $x > 0$ and like `ceiling()` for $x < 0$

```
> floor(1.6); floor(-1.6)
[1] 1
[1] -2
> ceiling(1.6); ceiling(-1.6)
[1] 2
[1] -1
> trunc(1.6); trunc(-1.6)
[1] 1
[1] -1
```


R trigonometric functions

Function	Description
`cos(x)`	cosine of `x` in radians
`sin(x)`	sine of `x` in radians
`tan(x)`	tangent of `x` in radians
`asin(x), acos(x), atan(x)`	inverse trigonometric functions for real or complex numbers
`asinh(x), acosh(x), atanh(x)`	inverse hyperbolic trigonometric functions for real or complex numbers

- all trigonometric functions measure angle in radians. R knows the value of π as `pi`

```
> pi
[1] 3.141593

> sin(pi/2)
[1] 1

> cos(pi/2)
[1] 6.123234e-17
```

R variable names and assignments

- variable names are case sensitive : `y` different from `Y`
- variable names must not begin with numbers (`4t`) or symbols (%8)
- variable names must not contain blank spaces (use `m.value` instead of `m value`)
- object assignment is achieved using the '<-', **gets arrow** operator. Do not put spaces between them or a logical test will be performed (see below)

```
> x <- 5
> x
[1] 5
> x < -5
[1] FALSE
```

NO SPACE for assignment

LOGICAL is 2 < -3 ?

- assignment can be achieved also with the '->', or '=' operators

```
> sqrt(x) + x^3 -> y
> y
[1] 127.2361
> z = x/y
> z
[1] 0.03929703
```


R arithmetic operators summary

+ - * /	sum, subtraction, multiplication, division
%/% %% ^	integer quotient, modulo, power
> >= < <= == !=	relational operators
! & |	logical not, and, or
~	model formulae ('is modelled as a function of')
<- ->	assignment (gets)
$	list indexing (the 'element name' operator)
:	sequence creation operator

```
> 119 %/% 12  # integer part of the division
[1] 9

> 119 %% 12   # reminder (modulo) of the division
[1] 11

> 15421 %% 7 == 0
[1] TRUE
```

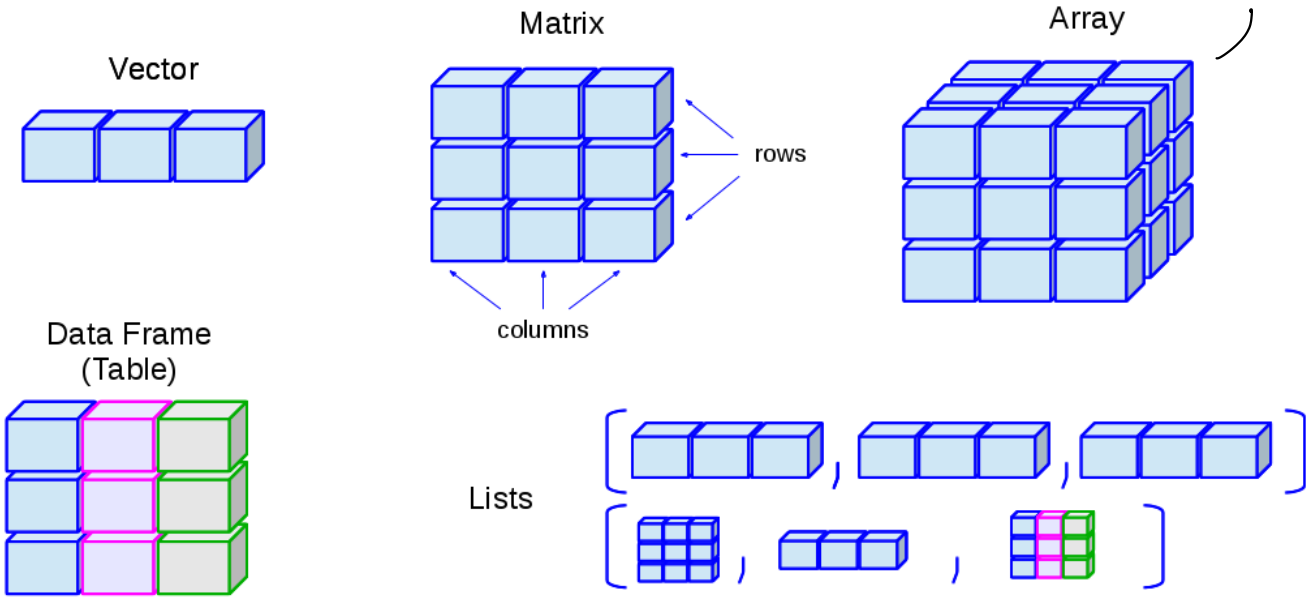
- several of these operators have different meaning inside model formulae :
  * indicates the main effects plus interaction (rather than multiplication),
  : the interaction between two variables (rather than generate a sequence), and
  ^ interactions up to the indicated power (rather than raise to the power)

R data types

- everything in R is an object
- the following data types are available:
- atomic data types: Vector (1-dim), Matrix (2-dim), Array ($>$ 2-dim)
- Data Frame: with homogeneous data type in each column
- List: a collection of simpler data types

all elements should be of the same time

multi dimension matrix



R data types: vectors

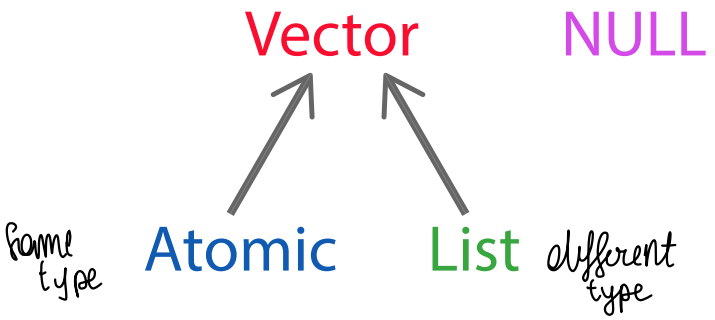
Alberto Garfagnini

Università di Padova

R lecture 2

R data types

- the most important family of data type is: `vector`
- all other data types are known as `nodes` (i.e. functions and environments)
- vectors can be:
  - atomic : all elements must have the same type
  - lists : elements can be of different types
  - `NULL` serves as generic zero length vector



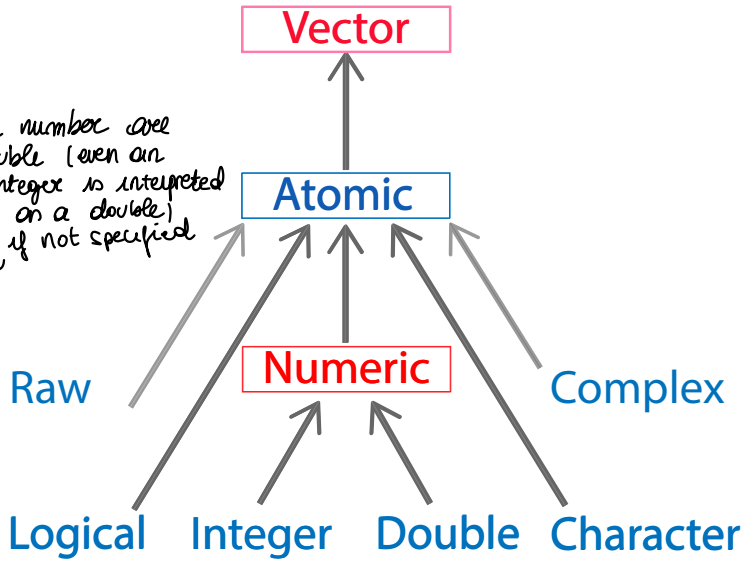
- every vector can have attributes
- two important attributes are: `dimension` and `class`
- `dimension` allows to create a matrix (`dim=2`) and array (`dim>2`)
- `class` powers the S3 object system in R

Atomic Vectors

R atomic vectors

- There are 4 basic types of atomic vectors:
  - logical : `TRUE, FALSE` abbreviated with `T` and `F` (logical value 1 or 0)
  - double : `2.75` (decimal), `1.23E4` (scientific) or `0xcafe` (hexadecimal) (2 →)
  - integer : written similar to double, but with an `L` suffix (`123L, 1E3L` or `0xcafeL`) (2L →)
  - character : `"a", "a word"`. These are strings surrounded by `"` or `'` (' → will be translated as ") special characters are escaped with \ See `?Quotes` for details

All number are double (even an integer is interpreted as a double) if not specified

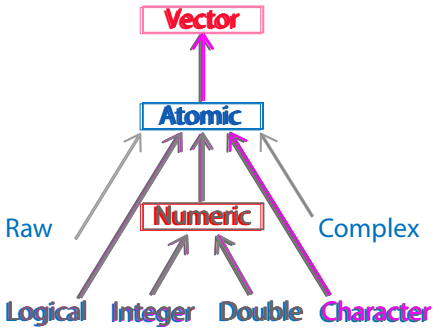


- and 2 rare types:
  - complex : `4.5 + 3i`
  - raw : (intended to hold raw bytes)

R atomic vectors

- R provides a set of functions to examine an object:

- `class()` : return the object class type
- `typeof()` : return the the object's data type
- `length()` : return the number of elements
- `attributes()` : return object metadata
- `str()` : display the internal structure of an R object




```
x <-3
class(x)
%> [1] "numeric"
typeof(x)
%> [1] "double"
length(x)
%> [1] 1
str(x)
%> num 3
```

type — how it's defined

```
y <- 3L
class(y)
%> [1] "integer"
typeof(y)
%> [1] "integer"
length(y)
%> [1] 1
str(y)
%>  int 3
```

```
z <- x>0
class(z)
%> [1] "logical"
typeof(z)
%> [1] "logical"
length(z)
%> [1] 1
str(z)
%>
logi true
```

```
w <- 'three'
class(w)
%> [1] "character"
typeof(w)
%> [1] "character"
length(w)
%> [1] 1
str(w)
%>
chr "three"
```


R vectors

How to create a vector → use function `c( , , )`

- scalar types do not exist, they are considered one-element vectors

```
x <-4.7; length(x)
%> [1] 1
```

- longer vectors are usually created with the concatenate, `c()`, function
- the size of a vector is determined at creation time

```
y <- c(1, 2, 5, 8)
str(y)  → structure
%>  num [1:4] 1 2 5 8
    type  index  shows first elements
```

- `c()` calls combined:

```
y <- c(y, 12, c(1, 7, 8))   concatenation
str(y)
%>  num [1:8] 1 2 5 8 12 1 7 8
```

Generating sequences of numbers

- an useful way to create a vector is to generate a sequence of numbers

```
0:10 # a sequence from 0 to 10, in steps of 1
%>  [1]  0  1  2  3  4  5  6  7  8  9 10

15:5 # a sequence from 15 down to 5
%>  [1] 15 14 13 12 11 10  9  8  7  6  5
```

- the seq() function allows to generate sequences in steps other than 1

```
seq(-2, 3, 0.5)
%>  [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0

seq(6, 4.2, -0.2)
%>  [1] 6.0 5.8 5.6 5.4 5.2 5.0 4.8 4.6 4.4 4.2
```

- or with a fixed vector length

```
seq(from=0.04, to=0.14, length=6)
%> [1] 0.04 0.06 0.08 0.10 0.12 0.14

seq(from=0.04, to=0.14, length=7)
%> [1] 0.04000000 0.05666667 0.07333333 0.09000000 0.10666667
%> [6] 0.12333333 0.14000000
```


Generating replicated values

- the function rep() replicates the values in a vector

```
rep (9,5) # replicate 5 times the number 9
%> [1] 9 9 9 9 9

rep(1:4, 2) # replicate twice the 1:4 sequence
%> [1] 1 2 3 4 1 2 3 4

rep(1:4, each=2) # replicate twice each sequence number
%> [1] 1 1 2 2 3 3 4 4

rep(1:4, each=2, times=3)
%>  [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4


# replicate each sequence number a different number of times
rep(1:4, 1:4)
%>  [1] 1 2 2 3 3 3 4 4 4 4

rep(c("cat","dog","mouse"), c(2,3,2))
%> [1] "cat"   "cat"   "dog"   "dog"   "dog"   "mouse" "mouse"
```

Infinity and 'not numbers'

It's actually implemented in R

- calculations can lead to results which go to $\pm\infty$ or are indeterminate

```
4/0
%> [1] Inf

-15/0
%> [1] -Inf
```

- but calculations involving $\pm\infty$ are properly evaluated

```
exp(-Inf)
%> [1] 0

exp(Inf)
%> [1] Inf

0/Inf
%> [1] 0

(0:3)^Inf
%> [1]   0   1 Inf Inf
```


Infinity and 'not numbers'

- some calculations may lead to results which are indeterminate, i.e. not numbers

```
0/0
%> [1] NaN
```

Not a number

```
Inf - Inf
%> [1] NaN

Inf/Inf
%> [1] NaN
```

- there are functions to test weather a number is finite or infinite

```
x <- -4.5
is.finite(x)
%> [1] TRUE

is.infinite(c(-4.5, 0/0, exp(Inf)))
%> [1] FALSE FALSE  TRUE

is.nan(c(-4.5, 0/0, exp(Inf)))
%> [1] FALSE  TRUE FALSE
```

Missing or unknown values

- R represents missing or unknown values with the sentinel NA
- but most computations with NA will return NA

```
NA > 0; 2.7*NA; ! NA
%> [1] NA
%> [1] NA
%> [1] NA
```

NOT AVAILABLE

If one number is not available it will return NA eg mean( ) → NA

⇒ mean(a1, na.rm = T)

- exception: when some identity holds for all possible inputs

```
NA ^ 0
%> [1] 1
NA | TRUE
%> [1] TRUE
NA & FALSE
%> [1] FALSE
```

- but, how do we check for NA values ?

```
y <- c(4, NA, -8)
y == NA  # it does not work, sets all to NA
%> [1] NA NA NA

y == "NA" # this does not work, either
%> [1] FALSE    NA FALSE

is.na(y) # this is the proper way
%> [1] FALSE  TRUE FALSE

y[! is.na(y)] # produce a vector with NAs removed
%> [1]  4 -8
```


Missing values: NA

- some built-in functions allow to skip NAs from computations

```
y <- c(4, NA, -8) ; mean(y)
%> [1] NA
mean(y, na.rm=TRUE)
%> [1] -2
```

or mean(a1[! is.na(a1)])

VECTOR INDICES STARTS at 1

comment

- how to we find the locations of NA values within a vector ?

```
vmv <- c(1:6, NA, NA, 8:12)
# Get the index of the values
seq(along=vmv)
%> [1]  1  2  3  4  5  6  7  8  9 10 11 12 13

seq(along=vmv)[is.na(vmv)] # and now of the NAs
%>[1] 7 8
```

When checking a vector for NA elements must use this function

insert negation

! is.na( )

check for valid

which(logic)

position of true going a logical vector

```
which(is.na(vmv)) # a simpler way exists
%>[1] 7 8
```

- if NAs are 'zero-count' values, we may want to replace them with 'zeros'

```
vmv[is.na(vmv)] <- 0 ; vmw
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12

vmv[which(is.na(vmv))] <- 0 ; vmw
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12
```

or

```
ifelse(is.na(vmv), 0, vmv) # use the 'vectorized' ifelse function
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12
```

vectorize function

WORK w VECTOR (NOT LOOP) is better

Vector indexing

- V [-1] remove last
- V [- lenght(V)]

– remove elements from vector

- the advantage of vector-based language is that it is simple to make computation involving all values in the vector

```
probe <- c(4, 7, 6, 5, 6, 7)
length(probe)
%> [1] 6
mean(probe)
%> [1] 5.833333
min(probe)
%> [1] 4
max(probe)
%> [1] 7
```

- subscripting is done through square brackets [] (indexing starts at '1')

```
index <- c(1, 3, 4, 6) # a vector of selected indexes
probe[index]
%> [1] 4 6 5 7
probe[c(1, 3, 4, 6)] # this is also valid
%> [1] 4 6 5 7
```


Vector indexing

- unwanted values can be dropped using negative indexes

```
probe <- c(4, 7, 6, 5, 6, 7) ; probe
%> [1] 4 7 6 5 6 7

probe[-1] # remove the first element
%> [1] 7 6 5 6 7

probe[-length(probe)] # remove the last element
%> [1] 4 7 6 5 6
```

- write a function to remove the smallest two values (with index 1 and 2) and largest two values (which will have subscripts `length(x)` and `length(x)-1`)

```
trim <- function(x) sort(x)[-c(1,2,length(x)-1,length(x))]
trim(probe)
%> [1] 6 6
```

name of function — variable — function called — [] selection — SORT → automatically remove NA

sort value so I can remove the smallest and largest that will have index 1,2 . Lenght(x) Lenght(x)-1

- sequences can be used to extract values

```
probe[1:3]
%> [1] 4 7 6
probe[seq(1,length(probe),2)]
%> [1] 4 6 6
probe[seq(1,length(probe),2)] # get odd indexes values
%> [1] 4 6 6
probe[seq(2,length(probe),2)] # get even indexes values
%> [1] 7 5 7
```

Vector attributes

- more complicated data structures, like matrices, arrays, factors and datetimes are built on top of vector by adding attributes
- attributes are also used to create user-defined S3 classes
- attributes are name/value pairs
- they can be retrieved/modified with `attr()` or retrieved en masse with `attributes()`

```
counts <- c(25,12,7,4,6,2,1,0,2)

attr(counts, "nx") <- "count1"
attr(counts, "ny") <- "events"

attr(counts, "nx")
%> [1] "count1"

attributes(counts)
%> $nx
%> [1] "count1"

%> $ny
%> [1] "events"
```

- most attributes are lost during operations, unless they are part of an S3 class
- only `names` and `dim` attributes are preserved

Vector attribute: `names`

- a vector can be given a name in three ways

```
% When creating it
x <- c(a = 1, b = 2, c = 3)

% By assiging a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")

% Inline, with the function setNames()
x<- setNames(1:3, c("a", "b", "c"))

x
%> a b c
%> 1 2 3
```

- vector names can be retrieved with the function `names()`

```
names(x)
%> [1] "a" "b" "c"
```

- and removed with `uname()`, or setting `names(x) <- NULL`

```
unname(x)
[1] 1 2 3
names(x) <- NULL
x
%> [1] 1 2 3
```

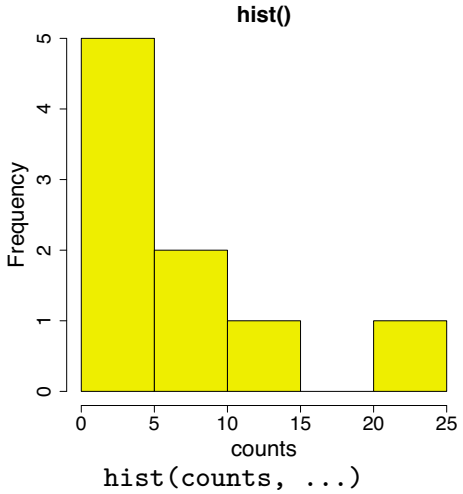
Example : naming vector elements

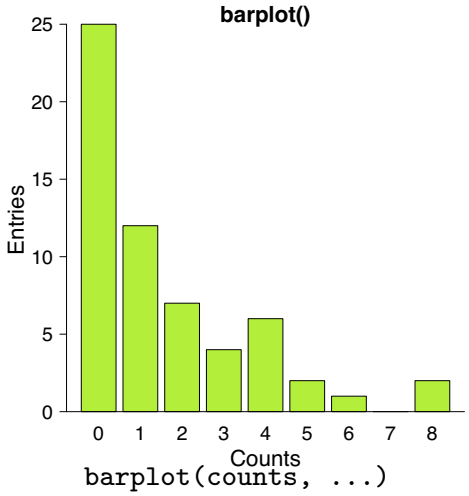
- sometimes is is useful to have values in a vector labelled
- for instance, we have a vector of counts occurrence of 0, 1, 2, ...

```
counts <- c(25,12,7,4,6,2,1,0,2)

names(counts) <- 0:(length(counts)-1)
str(counts)
%> Named num [1:9] 25 12 7 4 6 2 1 0 2
%> - attr(*, "names")= chr [1:9] "0" "1" "2" "3" ...

names(counts) <- NULL  # names can be easily removed
str(counts)
%> num [1:9] 25 12 7 4 6 2 1 0 2
```





Vector attribute: dimensions

- adding a `dim` attribute to a vector, changes it behavior to
- a 2D matrix `dim = c(nrow, ncol)`

```
v1 <- c(1:20)  ;   v1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

class(v1)
%> [1] "integer"

str(v1)
%> int [1:20] 1 2 3 4 5 6 7 8 9 10 ...

%% We transfor the vector to a matrix 4 x 5:
dim(v1) <- c(4,5)

class(v1)
%> [1] "matrix"

str(v1)
%> int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...

v1
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    5    9   13   17
%> [2,]    2    6   10   14   18
%> [3,]    3    7   11   15   19
%> [4,]    4    8   12   16   20
```

Vector attribute: dimensions

- adding a `dim` attribute to a vector, changes it behavior to
- a multi-dimensional array `dim = c(dim1, dim2, ... dimn)`

```
v1 <- c(1:20)

dim(v1) <- c(2,5,2)

class(v1)
%> [1] "array"

str(v1)
%> int [1:2, 1:5, 1:2] 1 2 3 4 5 6 7 8 9 10 ...

v1
%> , , 1
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    3    5    7    9
%> [2,]    2    4    6    8   10
%>
%> , , 2
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]   11   13   15   17   19
%> [2,]   12   14   16   18   20
```


Matrices and Arrays

- matrices and arrays can be created with the functions `matrix()` and `array`

```
v1 <- c(1:20)
matrix(v1, nrow=4, ncol=5)
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    5    9   13   17
%> [2,]    2    6   10   14   18
%> [3,]    3    7   11   15   19
%> [4,]    4    8   12   16   20

array(v1, c(2,5,2))
%> , , 1
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    3    5    7    9
%> [2,]    2    4    6    8   10
%>
%> , , 2
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]   11   13   15   17   19
%> [2,]   12   14   16   18   20
```

Vectors and logical subscripts

```
(x <- 0:10)
%> [1]  0  1  2  3  4  5  6  7  8  9 10

sum(x)
%>[1] 55

sum(x<5)
%>[1] 5
```

- the first `sum()` call sums up all the numbers in the vector
- the second call does not return the sum of the values which are lower than five

```
x<5
%> [1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

- `x<5` is a vector of logicals, but summing it up R converts logical `TRUE` to 1 and `FALSE` to 0
- we need vector subscripting to perform the desired sum

```
x[x<5]
%> [1] 0 1 2 3 4

sum(x[x<5])
%> [1] 10
```

correct way

sum(V2<5) → logical sum
6 6 true

R vector functions

Function	Description
`max(x)`	the maximum value in `x`
`min(x)`	the minimum value in `x`
`sum(x)`	the sum of all values in `x`
`mean(x)`	arithmetic average of the values in `x`
`median(x)`	median value in `x`
`range(x)`	a vector with `min(x)` and `max(x)`
`var(x)`	sample variance of `x`
`cor(x,y)`	correlation between `x` and `y` vectors
`sort(x)`	a sorted version of `x`
`rank(x)`	a vector with the ranks of the `x` values
`order(x)`	a vector with the permutations to sort `x` in asc order
`quantile(x)`	a vector with: minimum, lower quantile, median, upper quantile and maximum of `x`
`cumsum(x)`	a running sum of the vector elements
`cumprod(x)`	a running product of the vector elements
`cummax(x)`	a vector of non-decreasing numbers with the cumulative maxima
`cummin(x)`	a vector of non-decreasing numbers with the cumulative minima
`pmax(x, y, z)`	vector containing the maximum of `x`, `y` or `z` for each position
`pmin(x, y, z)`	vector containing the minimum of`x`, `y` or `z` for each position
`colMeans(x)`	column means of a dataframe or matrix
`colSums(x)`	column sums of a dataframe or matrix
`rowMeans(x)`	row means of a dataframe or matrix
`rowSums(x)`	row sums of a dataframe or matrix

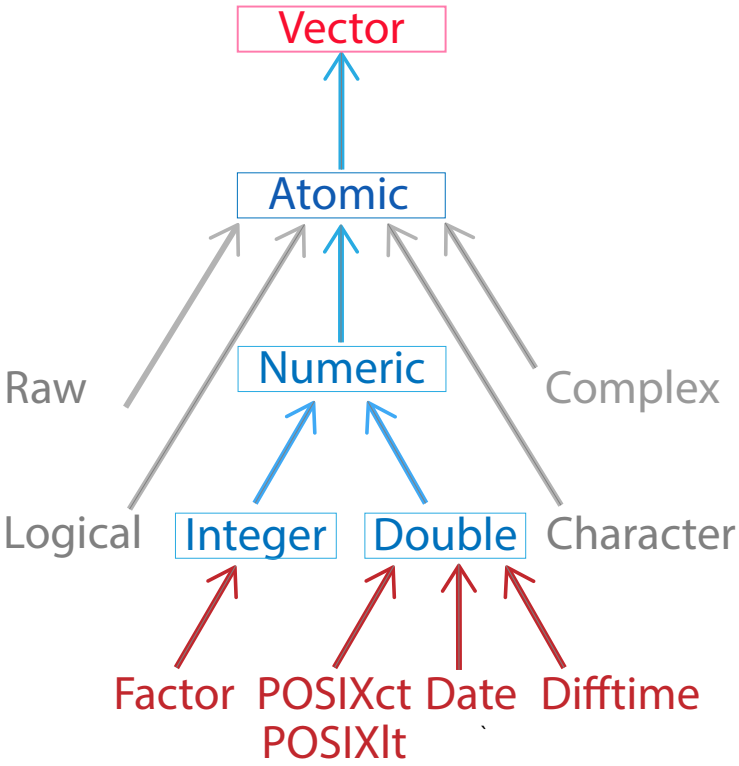
Null → empty

S3 Atomic Vectors

S3 atomic vectors

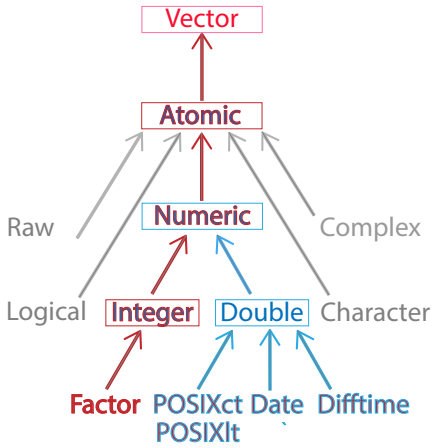
TYPE OF VECTORS

- S3 is the basic object system in R.
- an object is turned into an S3 object with a `class` attribute
- some important S3 vectors used in R are
  - `factor vectors` : used to store categorical data, as a fixed set of levels
  - `Date vectors`, for time object with day resolution
  - `POSIXct/POSIXlt vectors`, for time object with second (or sub-second) resolution
  - `difftime vectors`, for storing time durations



S3 atomic vectors : factors

- a `factor` is a vector that contains only predefined values
- it is used to store categorical data




```
x <- factor(c("a", "b", "b", "c"))
str(x)
%> Factor w/ 3 levels "a","b","c": 1 2 2 3

typeof(x)
%> [1] "integer"
attributes(x)
%> $levels
%> [1] "a" "b" "c"
%>
%> $class
[%> 1] "factor"
```

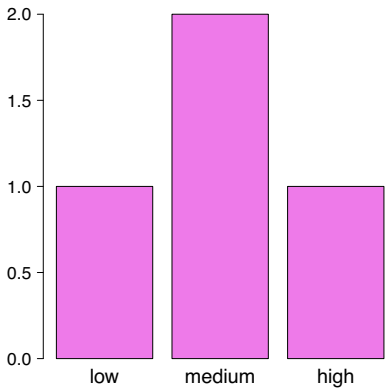
```
coord <- factor(c("Est", "West", "Est", "North"),
                levels = c("North", "Est", "South", "West")) ; coord
%> [1] Est   West  Est   North
%> Levels: North Est South West

> table(coord)
%> coord
%> North   Est South  West
%>     1     2     0     1
```


S3 atomic vectors : ordered factors

- they behave like factors, but the order of the levels is meaningful

```
grade <- ordered(c("high", "low", "medium",
                   "medium"),
          levels = c("low","medium","high"))
str(grade)
%> Ord.factor w/ 3 levels
%>                "low"<"medium"<..: 3 1 2 2
summary(grade)
%>    low medium   high
%>      1      2      1
barplot(table(grade), color="orchid2")
```



Note

- in base R factors are encountered very frequently:
- many base R functions (`read.csv()`, `data.frame()`) automatically convert character vectors to factors
- to suppress this behavior use `stringsAsFactors = FALSE`
- factors are built on top of integers, be careful when treating them like strings

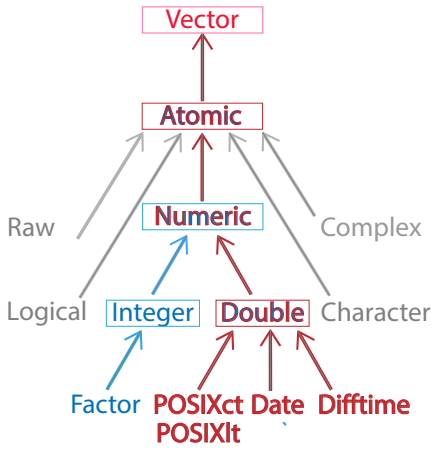
S3 atomic vectors : Dates

Date vectors are built on top of double vectors

```
today <- Sys.Date()  ;  today
%> [1] "2020-03-15"
typeof(today)
%> [1] "double"
class(today)
%> [1] "Date"

yesterday <- as.Date("2020-03-14")
yesterday
%> [1] "2020-03-14"

delta <- today - yesterday  ;  delta
%> Time difference of 1 days
class(delta)
%> [1] "difftime"
```



they are represented as number of days since 1970/01/01

```
days_since_1970_01_01 <- unclass(today)
days_since_1970_01_01
%> [1] 18336
class(days_since_1970_01_01)
%> numeric
```


S3 atomic vectors : Date-times

- baseR provides two ways of storing date-time information:

- POSIXct

  ct = calendar time (the time_t type in C)

- POSIXlt

  lt = local time (the struct tm type in C)

* POSIXct vectors are built on top of double vectors, and time is represented as seconds since 1970/01/01

```
now_ct <- as.POSIXct(Sys.time(), tzone="CET")
now_ct
%> [1] "2020-03-15 14:22:41 UTC"

r20bday_ct <- as.POSIXct("2020-02-29 12:00", tzone= "CET")
now_ct - r20bday_ct
%> Time difference of 15.14075 days
```

* the tzone attribute controls only how date-time is formatted, not how it is represented

```
structure(now_ct, tzone="Europe/Rome")
%> [1] "2020-03-15 15:30:53 CET"
structure(now_ct, tzone="Europe/Moscow")
%> [1] "2020-03-15 17:30:53 MSK"
structure(now_ct, tzone="Asia/Chongqing")
%> [1] "2020-03-15 22:30:53 CST"
```

Time duration

- durations represent the time difference between two pair of dates or date-times
- they are stored in `difftimes`
- this S3 class has a `unit` attribute that determines how the difference should be interpreted

```
one_week <- as.difftime(1, units="weeks")
attributes(one_week)
%> $class
%> [1] "difftime"
%>
%> $units
%> [1] "weeks"

today <- Sys.time()
next_sunday <- today + one_week

structure(next_sunday, tzone="Europe/Rome")
%> [1] "2020-03-22␣16:04:58␣CET"

fourty_min <- as.difftime(40, units="mins")
later <- today + fourty_min
later
%> [1] "2020-03-15␣15:44:58␣UTC"
```


`unique` and `duplicated` for vectors

- with the function `table()` we can inspect how many times each name appears
- the function `unique()` extracts the unique values in a vector, in the order in which the values are encountered in the vector

```
names <- c("John", "John", "Jim", "Anna", "Beatrix", "Anna")
table(names)
%> names
%>    Anna Beatrix      Jim     John
%>       2        1        1        2

unique(names)
%> [1] "John"    "Jim"     "Anna"    "Beatrix"
```

- the function `duplicated` creates a vector of logical values which is TRUE if that name has already appeared in the vector

```
duplicated(names)
%> [1] FALSE  TRUE FALSE FALSE FALSE  TRUE

names[!duplicated(names)]
%> [1] "John"    "Jim"     "Anna"    "Beatrix"
```

Operating on sets: union, intersect and setdiff

- given two sets, the union() function gives a set with all elements, but counting only once those common to both sets

```
setA <- c ("a", "b", "c", "d", "e")
setB <- c ("d", "e", "f", "g")

union(setA,setB)
%> [1] "a" "b" "c" "d" "e" "f" "g"
```

- intersection() gives ony the elements they have in common

```
intersect(setA, setB)
%> [1] "d" "e"
```

- the difference between the two sets is order-dependent

```
setdiff(setA, setB)
%> [1] "a" "b" "c"
setdiff(setB, setA)
%> [1] "f" "g"

setequal(setA, setA) # compare if the sets are equal
%> [1] TRUE
setequal(setA, setB)
%> [1] FALSE

setA %in% setB
%> [1] FALSE FALSE FALSE  TRUE  TRUE
setA[setA %in% setB]   # equal to intersect(setA, setB)
%> [1] "d" "e"
```

R data types: Lists

Alberto Garfagnini

Università di Padova

R lecture 3

R internals: variables and objects creation

- We create a vector with three values and assign it to a reference variable, `x`

```
x <- c(1,2,3)
```

- we now copy `x` to another variable `y`:

```
y <- x
```

- and modify one element of `y`

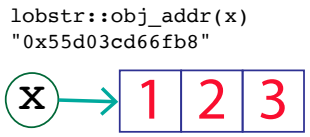
```
y[3] <- 4
```

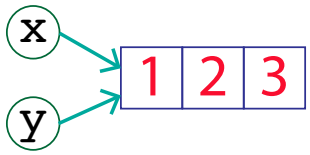
- did we modify also `x`?

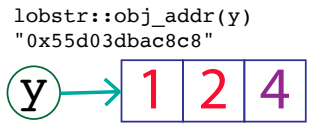
  No, they refer to two different objects:

```
str(x)
%> num [1:3] 1 2 3
str(y)
%> num [1:3] 1 2 4
```

- the behavior is called `copy-on-modify`
- all R objects are `immutable`







The `lobstr` package allows to visualize R data structures: it shows memory location and size of objects.
URL: `https://github.com/r-lib/lobstr`

R Lists

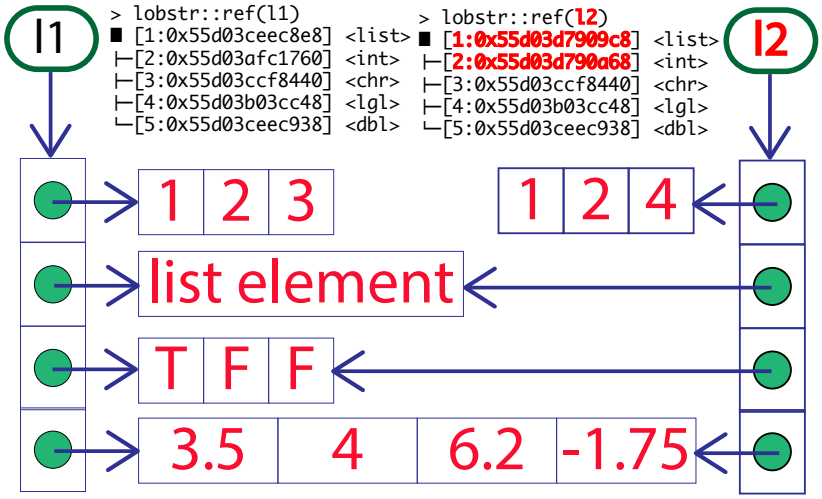
- Lists are an evolution of atomic vectors: each element can be of any type
- from the technical point of view: each element of a list is of the same type: it is a reference to another R object
- building a list:

```
l1 <- list( 1:3,
            "list element",
            c(TRUE, FALSE, FALSE),
            c(3.5, 4, 6.2, -1.75)
)
typeof(l1)
%> [1] "list"
```

- we copy to a new list and modify one element

```
l2 <- l1

l2[[1]] <- c(1L,2L,4L)
```



R matrices

- a matrix is a 2-dimensional object
- the first way of creating a matrix is by calling the `matrix()` object constructor

```
X <- matrix(c(1,0,0,0,1,0,0,0,1), nrow=3)   ;   X
%>      [,1] [,2] [,3]
%> [1,]    1    0    0
%> [2,]    0    1    0
%> [3,]    0    0    1

class(X)
%> [1] "matrix"
attributes(X)
%> $dim
%> [1] 3 3
str(X)
%> num [1:3, 1:3] 1 0 0 0 1 0 0 0 1
```

- another way is to transform a vector in a matrix: data can be arranged by rows (`byrow=T`) or columns (`byrow=F`)

```
vct <- c(1,2,3,4,4,3,2,1)
V <- matrix(vct,byrow=T,nrow=2)
V
%>      [,1] [,2] [,3] [,4]
%> [1,]    1    2    3    4
%> [2,]    4    3    2    1
```

```
V <- matrix(vct,byrow=F,nrow=2)
V
%>      [,1] [,2] [,3] [,4]
%> [1,]    1    3    4    2
%> [2,]    2    4    3    1
```

R matrices

- another possibility is to convert the vector to a matrix by specifying the new dimensions (rows and columns), using the `dim` function

```
vct <- c(1,2,3,4,4,3,2,1)
vct
%> [1] 1 2 3 4 4 3 2 1

dim(vct) <- c(4,2)
is.matrix(vct)
%> [1] TRUE

vct
%>      [,1] [,2]
%> [1,]    1    4
%> [2,]    2    3
%> [3,]    3    2
%> [4,]    4    1
```

- we can then transform the matrix:

```
tvct <- t(vct) # transpose the matrix
tvct
%>      [,1] [,2] [,3] [,4]
%> [1,]    1    2    3    4
%> [2,]    4    3    2    1
```


Accessing or operating on matrix rows or columns

- Let's create a matrix with $n = 20$ entries sampled from a Poisson distribution with $\lambda = 1.5$

```
X <- matrix(rpois(n=20,lambda=1.5), nrow=4)
X
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    1    1    2    4
%> [2,]    1    1    3    3    2
%> [3,]    1    3    5    0    1
%> [4,]    2    1    1    2    2

X[3,3] # return element in row 3 and column 3
%> [1] 5
X[4,] # return row 4
%> [1] 2 1 1 2 2
X[,5] # return column 5
%> [1] 4 2 1 2
```

- there are special functions for calculating summary statistics on a matrix:

```
rowSums(X)         # use colSums(X) for columns
%> [1]  9 10 10  8
rowMeans(X)        # use colMeans(X) for columns
%> [1] 1.8 2.0 2.0 1.6
```

Adding rows and columns to a matrix

- given a matrix, we would like to add a row, at the bottom, showing the column means, and a column at the right showing the row variances:

```
vct <- matrix(c(1,0,2,5,1,1,3,1,3,1,0,2,1,0,2,1), byrow=T, nrow=4)
vct
%>      [,1] [,2] [,3] [,4]
%> [1,]    1    0    2    5
%> [2,]    1    1    3    1
%> [3,]    3    1    0    2
%> [4,]    1    0    2    1
```


```
vct <- rbind(vct, apply(vct, 2, mean))
vct <- cbind(vct, apply(vct, 1, var))

colnames(vct) <- c(1:4, "variance")
rownames(vct) <- c(1:4, "mean")

vct
%>          1   2    3    4  variance
%> 1     1.0 0.0 2.00 5.00 4.6666667
%> 2     1.0 1.0 3.00 1.00 1.0000000
%> 3     3.0 1.0 0.00 2.00 1.6666667
%> 4     1.0 0.0 2.00 1.00 0.6666667
%> mean  1.5 0.5 1.75 2.25 0.5416667
```


The apply() collection functions

- are used to apply operations on a the elements of a complex object (vector, list, data.frame, ...) avoiding the use of loops
- apply() is the most basic and can be used over a matrix or array

apply()

- usage:

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

  X: an array or matrix

  MARGIN:  a value or range between 1 and 2
           to define where to apply the function:
  MARGIN=1: the manipulation is performed on rows
  MARGIN=2: the manipulation is performed on columns
  MARGIN=c(1,2) the manipulation is performed
                on rows and columns

  FUN: which function to apply.
       Built functions like mean, median, sum, min, max
       and user-defined functions can be applied
```

The apply() collection functions

- sapply() takes a vector or list object and returns an object of the same type.

sapply()

- usage:

```
sapply(X, FUN)

Arguments:
  X: A vector or an object
  FUN: Function applied to each element of X

x <- 1:10
apply(x, 1, sqrt)
#> Error in apply(x, 1, sqrt) : dim(X) must have a positive length

sapply(x, sqrt)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278
```


The apply() collection functions

- lapply() perfoms operations on list objects and returns a list object with the same length of the original set
- lappy() applies a function to each element of the list
- lapply() takes list, vector or data frame as input and returns a list

lapply()

- usage:

```
lapply(X, FUN)

Arguments:

X: A vector or an object
FUN: Function applied to each element of X

box <- c("Orange", "CHErrry", "APPLE")
str(box)
#> chr [1:3] "Orange" "CHErrry" "APPLE"

lbox <- lapply(box, tolower)
str(lbox)
#> List of 3
#>  $ : chr "orange"
#>  $ : chr "cherrry"
#>  $ : chr "apple"
```

apply(), sapply() and lapply()

- `apply()` is used to apply functions to rows or columns of matrices or dataframes across one of the margins of a matrix
- `margin=1` refers to the rows and `margin=2` to the columns

```
(Y <- matrix(rbinom(20, 9, 0.45), nrow=4))
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    6    5    4    2    5
%> [2,]    6    3    3    5    4
%> [3,]    5    2    3    4    4
%> [4,]    3    4    4    3    6
apply(Y, MARGIN=2, FUN=sum)  # apply sum() to all columns
%>[1] 20 14 14 14 19
```

- we can `apply()` functions to the individual elements of a matrix. In this case, the `margin` parameter, determines only the shape of the resulting matrix

```
apply(Y, 1, sqrt)
%>          [,1]     [,2]     [,3]     [,4]
%> [1,] 2.449490 2.449490 2.236068 1.732051
%> [2,] 2.236068 1.732051 1.414214 2.000000
%> [3,] 2.000000 1.732051 1.732051 2.000000
%> [4,] 1.414214 2.236068 2.000000 1.732051
%> [5,] 2.236068 2.000000 2.000000 2.449490
apply(Y, 2, sqrt)
%>          [,1]     [,2]     [,3]     [,4]     [,5]
%> [1,] 2.449490 2.236068 2.000000 1.414214 2.236068
%> [2,] 2.449490 1.732051 1.732051 2.236068 2.000000
%> [3,] 2.236068 1.414214 1.732051 2.000000 2.000000
%> [4,] 1.732051 2.000000 2.000000 1.732051 2.449490
```


apply(), sapply() and lapply()

- it is also possible to apply an anonymous, user defined, function

```
apply(Y, 1, function(x) x^2+x)  # compute x^2 + x for each element
%>      [,1] [,2] [,3] [,4]
%> [1,]   42   42   30   12
%> [2,]   30   12    6   20
%> [3,]   20   12   12   20
%> [4,]    6   30   20   12
%> [5,]   30   20   20   42
```

- in case you need to apply a function to a vector, rather than to the margin of a matrix, use `sapply()`

```
sapply(12:14, seq) # generate a list of seq, from 1:12 to 1:14
%> [[1]]
%>  [1]  1  2  3  4  5  6  7  8  9 10 11 12
%>
%> [[2]]
%>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
%>
%> [[3]]
%>  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

Random numbers

- random numbers from a uniform distribution $\mathcal{U}(0,1)$ are generated using `runif()`
- the random generation seed is set via `set.seed()`

```
set.seed(2019)
runif(3)
%> [1] 0.7699015 0.7128397 0.3033602
```

- resetting the seed with the same value will generate the same sequence of random numbers
- is is also possible to save the current seed and reuse it to obtain the same random numbers sub-sequence

```
current.seed <- .Random.seed  # save the current seed
runif(3)
%> [1] 0.61823636 0.05048374 0.04321880
runif(3)
%> [1] 0.820176206 0.009614496 0.102491504

current.seed -> .Random.seed # reset the previous sequence seed
runif(5)
%> [1] 0.618236361 0.050483740 0.043218804 0.820176206 0.009614496
```


sampling from a vector

- generating random numbers from probability distributions will be discussed in the next lessons
- now we want to randomize (shuffling or sampling from) the elements of a vector
- There are two ways of sampling:
  1) sampling without replacement : all the vector values will appear in output, but in a randomized sequence
  2) sampling with replacement : some vector values may be re-selected and appear more than once in the output
- using `sample()`, sampling without replacement is the default operation

```
y <- c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
sample(y) # reshuffling all vector values
%>  [1]  3  9  2  5  4  8  6  8  6  4 10  3  7 11  9
sample(y, 5) # pick up only 5 values from the original vector
%> [1] 3 8 9 4 8
sample(y, 5) # just redo it, and a different sequence may appear
%> [1] 8 3 8 4 3
```

- The option `replace=T` allows for sampling with replacement

```
sample(y, replace=T)
%>  [1]  8  3  6  8  8  4  3  7 10  9 10  9  4  4  7
```

sample()'s surprise example

```
x <- 1:10
x
%>  [1]  1  2  3  4  5  6  7  8  9 10

sample(x[x>8])
%> [1] 10  9

sample(x[x>9])
%>  [1]  1 10  8  7  6  5  4  2  9  3

sample(x[x > 10])
%> integer(0)
```

```
sample(x, size, replace = FALSE, prob = NULL)
If 'x' has length 1, sampling takes place from '1:x'
```

- the first argument of `sample()` can be a vector of more than one element or an integer
- the `resample()` function is safer

```
sample(15)
%>  [1]  2  4  3 11  7 14  6  5  1 13 15 10 12  9  8

str(x[x>9])
%>  int 10

resample(x[x>8])
%> [1] 10  9
resample(x[x>9])
%> [1] 10
```

the `resample()` function is available in the `gdata` package. https://cran.r-project.org/web/packages/gdata/index.html

R subsetting

- R's subsetting operators are fast and powerful, and allow to perform complex operations in a way that few other languages can match

- there are 6 ways to subset atomic vectors
- there are 3 subsetting operators: `[[`, `[` and `$`
- subsetting can be combined with assignment

Subsetting atomic vectors - 1

```
x  <- c(2.1, 4, 6.7, 1.75)
```

- positive integers return elements at a specified position

```
x[c(1,3)]
%> [1] 2.1 6.7

% Duplicate indices will duplicate values
x[c(1,1,3,3)]
%> [1] 2.1 2.1 6.7 6.7

% Real numbers are truncated to integers
x[sort(x)]
%> [1] 2.10 4.00 1.75   NA
```

- negative integers exclude elements

```
x[-c(1,3)]
%> [1] 4.00 1.75

% NB negative and positive ints cannot be mixed
x[c(-1,3)]
%> Error in x[c(-1, 3)]:only 0's␣may␣be␣mixed␣with␣negative␣subscripts
```


Subsetting atomic vectors - 2

```
x  <- c(2.1, 4, 6.7, 1.75)
```

- logical vectors select elements where the logical value is TRUE

```
x[c(T, T, F, T)]
%> [1] 2.10 4.00 1.75

x[x>2]
%> [1] 2.1 4.0 6.7
```

- if in `x[sel]`, `length(sel) != length(x)` the recycling rules are used: the shorter vector is recycled to the length of the longer

```
> x[c(TRUE, FALSE)]
[1] 2.1 6.7

%# is equivalent to:
> x[c(TRUE, FALSE, TRUE, FALSE)]
[1] 2.1 6.7
```

- nothing returns the original vector

```
x[]
%> [1] 2.10 4.00 6.70 1.75
```

Subsetting atomic vectors - 3

```
x  <- c(2.1, 4, 6.7, 1.75)
```

- zero returns a zero-length vector (it can be helpful to generate test data)

```
x[0]
numeric(0)
```

- named vectors can be accessed with character vectors

```
y <- setNames(x, LETTERS[1:length(x)])
y
%>    A    B    C    D
%> 2.10 4.00 6.70 1.75
y["A"]
%>   A
%> 2.1

y[c('A', 'A', 'D')]
%>   A    A    D
%>2.10 2.10 1.75
```

- WARNING: subsetting with factors will use the underlying integer vector, not the character levels. ➜ Avoid subsetting with factors

```
y[factor("B")]
%>   A
%> 2.1
```


Subsetting matrices

- subsetting a matrix or a list works in a similar way as subsetting atomic vectors

```
S  <- matrix(1:9, nrow = 3)
%> [1,]  1  4  7
%> [2,]  2  5  8
%> [3,]  3  6  9
```

- using [] always returns a list
- [[]] and $ allows to pull out elements from the list
- the common rule to subset a matrix (2D) and an array (nD, $n > 2$) is to supply a 1D vector for each dimension, separated by a comma
- blank subsetting allows to keep all data for the corresponding dimension

```
%# Get rows 1 and 3 and all columns
S[c(1,3), ]
%>      [,1] [,2] [,3]
%> [1,]    1    4    7
%> [2,]    3    6    9

colnames(S) <- c("S1", "S2", "S3")
S[c(T, F, T), c("S1", "S3")]
%>      S1 S3
%> [1,]  1  7
%> [2,]  3  9
```

Subsetting matrices - 2

- matrices and arrays are just vectors with special attributes, therefore they can be subset with a single vector, as if they were a 1D vector

```
v <- outer(1:5,1:5, FUN="paste", sep=",")
v
%>      [,1]  [,2]  [,3]  [,4]  [,5]
%> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
%> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
%> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
%> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
%> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"

v[seq(3, 23, 5)]
%> [1] "3,1" "3,2" "3,3" "3,4" "3,5"
```

- to preserve the original matrix dimension, use `drop = FALSE`

```
(S <- matrix(1:6, nrow = 2))
%>      [,1] [,2] [,3]
%> [1,]    1    3    5
%> [2,]    2    4    6

S[1, ]
%> [1] 1 3 5

S[1, , drop = FALSE]
%>      [,1] [,2] [,3]
%> [1,]    1    3    5
```


Selecting a single element

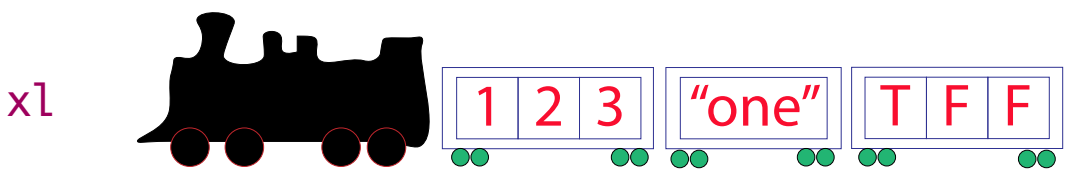
- there are two other subsetting operators:
  - `[[]]` is used to extract single items
  - `$` is used as a shorthand: `x$y` stands for `x[["y"]]`
- `[[]]` is most important while working with lists: subsetting a list with single `[]` always returns a smaller list

If list `xl` is a train carrying objects, then `xl[[5]]` is the object in car 5; `xl[4:6]` is a train of cars 4-6

https://twitter.com/RLangTip/status/268375867468681216

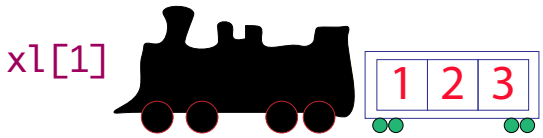
- with this metaphor let's build a list

```
xl <- list(1:3, "one", c(T,F,F))
```

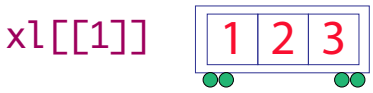


Selecting a single element

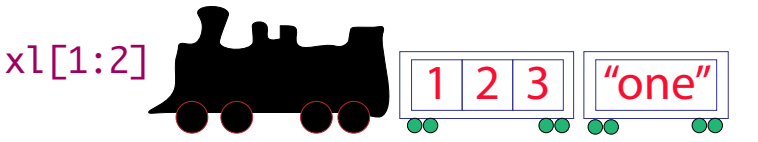
- two options are available when extracting a single element:
- create a smaller train, with fewer cars (using `[]`)

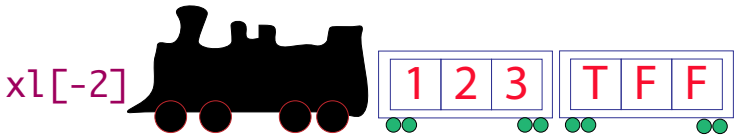


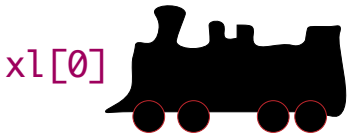
- or extract the content of a particular car (with `[[]]`)



- extracting multiple (or zero) elements, we have to build a smaller train







Loops

- Let's create a function, using loops, to evaluate the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1$

```
fac1 <- function(x) {
   f <- 1
   if (x<2) return (1)
   for (i in 2:x) {
       f <- f*i
   }
   return (f)
}
sapply(1:5, fac1)
%> [1]   1   2   6  24 120
```

```
fac2 <- function(x) {
   f <- 1; t <- x
   while (t>1) {
       f <- f*t
       t <- t-1
   }
   return(f)
}
sapply(1:5, fac2)
%> [1]   1   2   6  24 120
```

```
fac3 <- function(x) {
   f <- 1; t <- x
   repeat {
       if (t<2) break
       f <- f*t
       t <- t-1
   }
   return(f)
}
sapply(1:5, fac3)
%> [1]   1   2   6  24 120
```

- But it is almost always better to use a built-in function that operates on the entire vector, removing the need of loops or repeats

```
> cumprod(1:5) # it does not work for 0
[1]   1   2   6  24 120
> fac4 <- function(x) max(cumprod(1:x))
> sapply(1:5, fac4)
[1]   1   2   6  24 120

> # R implements a factorial() function, introduced not long ago
> sapply(1:5,factorial)
[1]   1   2   6  24 120
```

Loop avoidance: use vectorized operations

- it's a good R programming practice to avoid loops wherever possible
- in many cases, using vector functions, makes it particularly straightforward

```
> y <- c(-3,4,-2,-1,8,7,9)
> y
[1] -3  4 -2 -1  8  7  9

> for (i in 1:length(y)) {if (y[i] < 0) y[i] <- 0}
> y
[1] 0 4 0 0 8 7 9
```

- in the example below, a loop can be replaced by logical subscripts

```
> y <- c(-3,4,-2,-1,8,7,9)

> y[y<0] <- 0

> y
[1] 0 4 0 0 8 7 9
```


the `ifelse()` vectorized function

- `ifelse()` allow to work on an entire vector without using loops
-

```
> y <- log(rpois(20,1.5))
> y
 [1]       -Inf 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
 [7] 0.6931472       -Inf 1.3862944 0.6931472 1.3862944       -Inf
[13]       -Inf 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
[19]       -Inf 1.0986123

> mean(y)
[1] -Inf

> (y <- ifelse(y<0, NA, y))
 [1]         NA 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
 [7] 0.6931472         NA 1.3862944 0.6931472 1.3862944         NA
[13]         NA 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
[19]         NA 1.0986123

> mean(y, na.rm=TRUE)
[1] 0.5431911
```

Loops are slow, compared to vectorized operations

- let's generate $5 \cdot 10^7$ events according to an uniform distribution, $\mathcal{U}(0,1)$
- we want to search for the maximum value in the vector using the vectorized function `max()` e by using conventional loops

```
x <- runif(50000000)

system.time(max(x))
%>   user  system elapsed
%>  0.106   0.000   0.106

pc <- proc.time()
cmax <- x[1]
for (i in 2:length(x)) { if(x[i]>cmax) cmax <- x[i] }

proc.time()-pc
%>   user  system elapsed
%>  2.061   0.071   2.133
```

- `system.time()` and `proc.time()` produce a vector of three numbers, showing the user, system and total elapsed time in seconds

Good/Bad practice in building vectors

- we want to build a vector containing 10^n elements in the sequence $1{:}10^n$
- three ways are analyzed

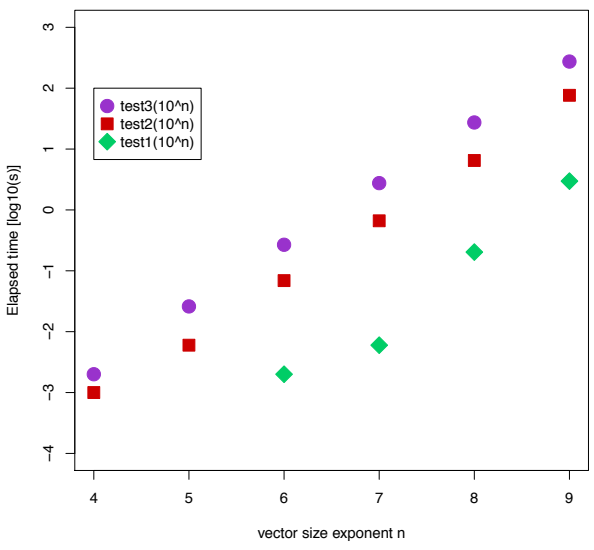
```
test1 <- function(n){
    y <- 1:n
}
```

```
test2 <- function(n){
    y <- numeric(n)
    for (i in 1:n)
        y[i] <- i
}
```

```
test3 <- function(n){
    y <- NULL
    for (i in 1:n)
        y <- c(y,i)
}
```

```
> system.time(test1(10000000))
   user  system elapsed
  0.006   0.000   0.006
> system.time(test2(10000000))
   user  system elapsed
  0.622   0.011   0.633
> system.time(test3(10000000))
   user  system elapsed
  2.755   0.003   2.758
```

- the first method (`test1`) is the best
- the loop using a pre-determined vector length is reasonably fast
- the last method (`test3`) is the slowest
- Moral: never grow vectors by repeated concatenation



R Lists example

- let's create a more complex list with etherogeneous object types

```
apples  <- c(4, 4.5, 5, 3.9)
oranges <- c(TRUE, FALSE, TRUE)
chalk   <- c("limestone", "marl", "ooline", "CaCO3")
pears   <- c(3.2-4.5i, 12.8+2.2i)

items <- list(apples, oranges, chalk, pears)
items
%> [[1]]
%> [1] 4.0 4.5 5.0 3.9
%>
%> [[2]]
%> [1]  TRUE FALSE  TRUE
%>
%> [[3]]
%> [1] "limestone" "marl"      "ooline"    "CaCO3"
%>
%> [[4]]
%> [1]  3.2-4.5i 12.8+2.2i
```


R List example: element access

- vectors, matrices and arrays subscripts have one set of square brackets [6], [3,4] or [2,3,2,1]
- lists subscripts have double square brackets [[2]] or [[i,j]]

```
items[[3]]
%> [1] "limestone" "marl"      "ooline"    "CaCO3"

items[[3]][1]
%> [1] "limestone"
```

- if the list elements have names, it is possible to use the operator $ for list indexing

```
names(items) <- c("apples", "oranges", "chalk", "pears")

str(items)
%> List of 4
%>  $ apples : num [1:4] 4 4.5 5 3.9
%>  $ oranges: logi [1:3] TRUE FALSE TRUE
%>  $ chalk  : chr [1:4] "limestone" "marl" "ooline" "CaCO3"
%>  $ pears  : cplx [1:2] 3.2-4.5i 12.8+2.2i

items$pears
%> [1]  3.2-4.5i 12.8+2.2i
```

R list example: Applying functions

- the length of the list is the number of items on the list. To get the length of the individual vectors we use the `lapply()` function

```
length(items)
%> [1] 4

lapply(items, length)
%> $apples
%> [1] 4

%> $oranges
%> [1] 3

%> $chalk
%> [1] 4

%> $pears
%> [1] 2
```

```
class(items)
%> [1] "list"

lapply(items, class)
%> $apples
%> [1] "numeric"

%> $oranges
%> [1] "logical"

%> $chalk
%> [1] "character"

%> $pears
%> [1] "complex"
```


R Lists : 3

- applying numeric functions to the list, will only work for objects of class `numeric` or `complex`

```
mean(items)
%> [1] NA
%> Warning message:
%> In mean.default(items) :
%>   argument is not numeric or logical: returning NA

lapply(items, mean)
%> $apples
%> [1] 4.35

%> $oranges
%> [1] 0.6666667

%> $chalk
%> [1] NA

%> $pears
%> [1] 8-1.15i
%> Warning message:
%> In mean.default(X[[i]], ...) :
%>   argument is not numeric or logical: returning NA
```

- a warning message points out that the third vector cannot be coerced to a number (it is not numeric, complex or logical), therefore NA appears in the output

R Lists : 4

- The `summary()` function works for lists, but the most useful overview of a list content is given by `str()`, the structure function:

```
summary(items)
%>         Length Class  Mode
%> apples  4      -none- numeric
%> oranges 3      -none- logical
%> chalk   4      -none- character
%> <NA>    2      -none- complex

str(items)
%> List of 4
%>  $ apples : num [1:4] 4 4.5 5 3.9
%>  $ oranges: logi [1:3] TRUE FALSE TRUE
%>  $ chalk  : chr [1:4] "limestone" "marl" "ooline" "CaCO3"
%>  $ NA     : cplx [1:2] 3.2-4.5i 12.8+2.2i
```


Question Time

- What is the effect of `[[1:2]]` on a list ?

```
xl <- list(1:3,
           "one",
           c(T, F, F))

% xl[[1:2]] is equivalent to xl[[1]][[2]]
xl[[1:2]]
[1] 2
xl[[1]][[2]]
%> [1] 2
% i.e. it extracts the element stored in the list at position 1, and
%      then it gets the second element

% This fails:
 xl[[1:3]]
%> Error in xl[[1:3]] : recursive indexing failed at level 2
%  and it is equivalent to:
> (a1 <- xl[[1]])
[1] 1 2 3
> (a2 <- a1[[2]])
[1] 2
> (a3 <- a2[[3]])
Error in a2[[3]] : subscript out of bounds
```

R data types: Data Frames

Alberto Garfagnini

Università di Padova

R lecture 4

R Data frames

- two important S3 vectors built on top of lists are `data frames` and `tibbles`
- a `data frame` is like a matrix, with a 2-dim rows-and-columns structure
- it's a named list of vectors, with attributes for columns and rows names, (`names`, `row.names`), belonging to the `data.frame` class

- technically, a `data frame` is a `list` with all equal length vectors

```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
%> [1] "list"

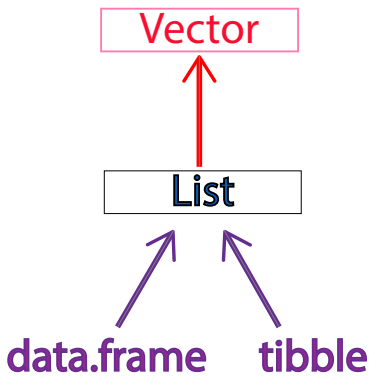
attributes(df1)
%> $names
%> [1] "x" "y"

%> $class
%> [1] "data.frame"

%> $row.names
%> [1] 1 2 3

str(df1)
%> 'data.frame':        3 obs. of  2 variables:
%>  $ x: int  1 2 3
%>  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Now dataframe reads correctly the type (eg with letters)
letters ⟶ now chr
--→ before FACTOR



R Data Frames : examples

- we have a table with the results of two exams for the stdent of an hipotetical course, and we want to import them in a `data.frame`

$Exam_1$	$Exam_2$	Gender
27	25	M
28	30	F
...		
27	27	M
25	28	F

```
exam1 <- c(27,28,24,24,30,26,23,23,24,28,27,25)
exam2 <- c(25,30,26,24,30,30,25,25,30,28,27,28)
gender <- c("M","F","M","M","M","M","M","M","F","F","M","F")

dc <- data.frame(exam1, exam2, gender)
head(dc, n=2) # extract the first two lines of the data frame
%>   exam1 exam2 gender
%> 1    27    25      M
%> 2    28    30      F
```

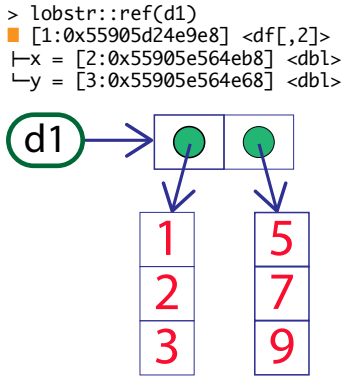
From R 4.0 `stringsAsFactors = FALSE` by default

```
dc1 <- data.frame(exam1, exam2, gender,
                  stringsAsFactors = FALSE)
str(dc1)
'data.frame':   12 obs. of  3 variables:
 $ exam1 : num  27 28 24 24 30 26 23 23 24 28 ...
 $ exam2 : num  25 30 26 24 30 30 25 25 30 28 ...
 $ gender: chr  "M" "F" "M" "M" ...
```


R Data Frames objects creation

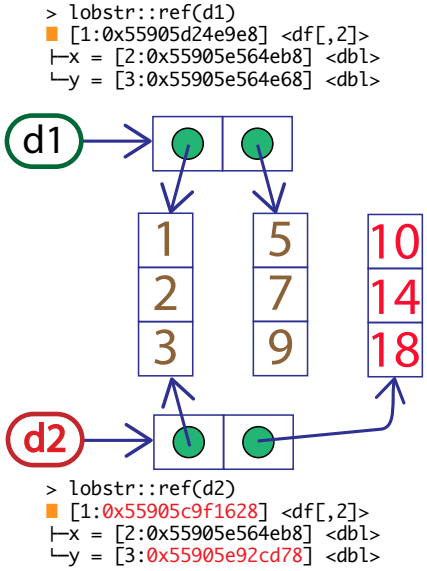
- Data frames are list of vectors, therefore copy-on-modify has important consequences

```
d1 <- data.frame(x = c(1, 2, 3),
                 y = c(5, 7, 9))
d1
%>   x y
%> 1 1 5
%> 2 2 7
%> 3 3 9
```



- if we modify a column ➜ only the reference to the new column will be updated

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
d2
%>   x  y
%> 1 1 10
%> 2 2 14
%> 3 3 18
```

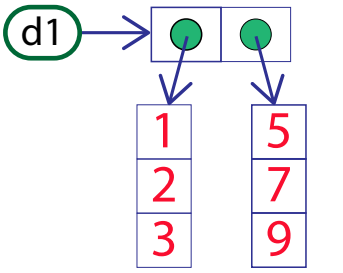


R Data Frames objects creation

- Data frames are list of vectors, therefore copy-on-modify has important consequences

```
d1 <- data.frame(x = c(1, 2, 3),
                 y = c(5, 7, 9))
d1
%>   x y
%> 1 1 5
%> 2 2 7
%> 3 3 9
```

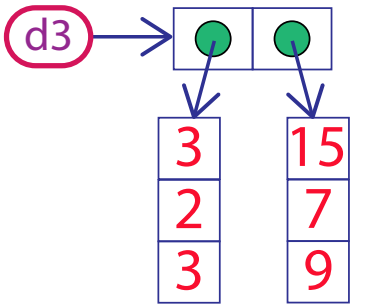
```
> lobstr::ref(d1)
■ [1:0x55905d24e9e8] <df[,2]>
├─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```



- but if any row is modified ➜ every column is modified because every column must be copied

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
d3
%>   x  y
%> 1 3 15
%> 2 2  7
%> 3 3  9
```

```
> lobstr::ref(d3)
■ [1:0x55905e6f1058] <df[,2]>
├─x = [2:0x55905ea0c238] <dbl>
└─y = [3:0x55905ea0c1e8] <dbl>
```

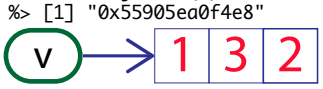


Modify-in-place

- Modifying an R object usually creats a copy
- but there are 2 exceptions:
  - objects with single binding get a special performance optimization
  - environments, a special type of object, are always modified in place

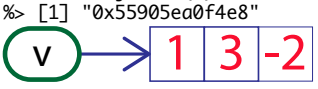
```
v <- c(1, 3, 2)
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```

```
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```




```
v[[3]] <- -2
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```

```
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```



- but it is very difficult to predicts when R applies this optimization
- concerning object binding, R only counts 0, 1 or `MANY`
- it means that if an object has 2 bindings (i.e. `many`), and one gets deleted, the reference does not go back to 1 (`many - 1 = many`)
- when a funtion is called, it makes a reference to the object ➜ it is very difficult to predict weather or not a copy will occur
- cfr: `https://developer.r-project.org/Refcnt.html`

Accessing `data frames` elements

- a `data frame` is a `list`, therefore we can access them via component index value `[[ j ]]` or via component names

```
str(dc)
%> 'data.frame':    12 obs. of  3 variables:
%>  $ exam1 : num  27 28 24 24 30 26 23 23 24 28 ...
%>  $ exam2 : num  25 30 26 24 30 30 25 25 30 28 ...
%>  $ gender: Factor w/ 2 levels "F","M": 2 1 2 2 2 2 2 2 1 1 ...

dc[[1]] # access by component index
%>  [1] 27 28 24 24 30 26 23 23 24 28 27 25

dc$exam1 # access by component name
%>  [1] 27 28 24 24 30 26 23 23 24 28 27 25
%> Levels: F M
```

- but a `data frame` can be treated in a matrix-like fashion, as well

```
dc[,1] # select column 1
%>  [1] 27 28 24 24 30 26 23 23 24 28 27 25

dc[1,1] # and access the single element, as well
%> [1] 27
```


Data frames row names

- data frames allow to label each row with a name, a character vector containing only unique names

```
df1 <- data.frame( age = c(35, 25, 18),
                   hair = c("blond", "brown", NA),
                   row.names = c("Bob", "Tom", "Sam"))
df1
%>      age  hair
%> Bob  35 blond
%> Tom  25 brown
%> Sam  18  <NA>

names(df1)
%> [1] "age"  "hair"

row.names(df1)
%> [1] "Bob" "Tom" "Sam"
```

- but row names are a bad practice:

(1) metadata is metadata : storing it in a different way to the rest of data is a bad idea

(2) row names are a poor abstraction for labeling rows : they only work when a row can be identified by a single string

(3) row names must be unique : any duplication of rows will create new row names ➜ complicated "string surgery" may be needed

Advanced data frames : data selection

```
dc[2:4,]  # Select only rows 2:4
%>   exam1 exam2 gender
%> 2    28    30      F
%> 3    24    26      M
%> 4    24    24      M

dc[-(2:10),] # drop rows 2:10
%>    exam1 exam2 gender
%> 1     27    25      M
%> 11    27    27      M
%> 12    25    28      F
```

- with the sample function , data can be selected at random

```
dc[sample(1:12,3),]  # select 3 rows at random
%>   exam1 exam2 gender
%> 8    23    25      M
%> 9    24    30      F
%> 6    26    30      M

dc[sample(1:12,3),]  # select 3 rows at random
%>    exam1 exam2 gender
%> 1     27    25      M
%> 10    28    28      F
%> 2     28    30      F
```


Advanced data frames : data selection

- suppose we want to extract all columns that contain numbers, rather than characters or logicals, from a data frame

```
dc[,sapply(dc,is.numeric)]
%>    exam1 exam2
%> 1     27    25
%> 2     28    30
%> 3     24    26
%> 4     24    24
%> 5     30    30
%> 6     26    30
%> 7     23    25
%> 8     23    25
%> 9     24    30
%> 10    28    28
%> 11    27    27
%> 12    25    28
```

```
dc <- data.frame(exam1, exam2, gender)
str(dc)
'data.frame':  12 obs. of  3 variables:
$ exam1 : num  27 28 ...
$ exam2 : num  25 30 ...
$ gender: Fact w/ 2 levels "F","M": 2 1
```

- and now we want to get only factors (and remove numerics)

```
dc[,sapply(dc,is.factor)]
%>  [1] M F M M M M M M F F M F
%> Levels: F M
```

Advanced data frames and NA elements

- sometimes our data frame can have missing values (NA) and we may need to omit those values
- we can create a shorter data frame using the na.omit() function

```
data
%>    slope   pH area
%> 1     11 4.1  3.6
%> 2     NA 5.2  5.1
%> 3      3 4.9  2.8
%> 4      5  NA  3.7
```

```
na.omit(data)
%>    slope   pH area
%> 1     11 4.1  3.6
%> 3      3 4.9  2.8
```

```
clean_data <- na.exclude(data)
clean_data
%>    slope   pH area
%> 1     11 4.1  3.6
%> 3      3 4.9  2.8
```

```
lapply(clean_data, mean)
%> $slope
%> [1] 7
%> $pH
%> [1] 4.5
%> $area
%> [1] 3.2
```

```
# Let's count the missing values
apply(apply(data,2,is.na),2,sum)
%> slope    pH  area
%>     1     1     0
```


Advanced data frames : sorting elements

```
dc[order(exam1),]
%>    exam1 exam2 gender
%> 7     23    25      M
%> 8     23    25      M
...
```

```
dc[order(exam1, decreasing=TRUE),]
%>    exam1 exam2 gender
%> 5     30    30      M
%> 2     28    30      F
```

- ```
  dc[order(gender, exam1, exam2, decreasing=TRUE),]
  %>    exam1 exam2 gender
  %> 5     30    30      M
  %> 11    27    27      M
  %> 1     27    25      M
  %> 6     26    30      M
  %> 3     24    26      M
  %> 4     24    24      M
  %> 7     23    25      M
  %> 8     23    25      M
  %> 2     28    30      F
  %> 10    28    28      F
  %> 12    25    28      F
  %> 9     24    30      F
  ```
  ```

Summary of data selection in `data frames`

- given a `data frame` called `data`, we assume `n` is a row number, and `m` is one of the column.
- the syntax `[n,]` selects all the columns given row `n`, while `[,m]` selects all the rows with column `m`

command	meaning
`data[n,]`	select all of the columns from row n of the data frame
`data[-n,]`	drop the whole of row n from the data frame
`data[1:n,]`	select all of the columns from rows 1 to n of the data frame
`data[-(1:n),]`	drop all of the columns from rows 1 to n of the data frame
`data[c(i,j,k),]`	select all of the columns from rows i, j, and k of the data frame
`data[x > y,]`	use a logical test ($x > y$) to select all columns from certain rows
`data[,m]`	select all of the rows from column m of the data frame
`data[,-m]`	drop the whole of column m from the data frame
`data[,1:m]`	select all of the rows from columns 1 to m of the data frame
`data[,-(1:m)]`	drop all of the rows from columns 1 to m of the data frame
`data[,c(i,j,k)]`	select all of the rows from columns i, j, and k of the data frame
`data[,x > y]`	use a logical test ($x > y$) to select all rows from certain columns
`data[,c(1:m,i,j,k)]`	add duplicate copies of columns i, j, and k to the data frame
`data[x > y,a != b]`	extract certain rows ($x > y$) and certain columns ($a! = b$)
`data[c(1:n,i,j,k),]`	add duplicate copies of rows i, j, and k to the data frame

The `tibble` data structure

- it is a modern reimagining of the `data frame`
- it is provided by the `tibble` package which is part of the `tidyverse` core library

```
library(tidyverse)
%>  Attaching packages --------  tidyverse 1.3.0
%>  ggplot2 3.3.0      purrr   0.3.3
%>  tibble  2.1.3      dplyr   0.8.5
%>  tidyr   1.0.2      stringr 1.4.0
%>  readr   1.3.1      forcats 0.5.0
%>  Conflicts ------------ tidyverse_conflicts()
%>  dplyr::filter() masks stats::filter()
%>  dplyr::lag()    masks stats::lag()
```

- a `data frame` can be converted to a `tibble`

```
dct <- tibble(dc)
dct
%> # A tibble: 12 x 1
%>    dc$exam1 $exam2 $gender
%>       <dbl>  <dbl> <fct>
%>  1       27     25 M
 ...
%> 12       25     28 F
```

tibble :: tibble() to load it
library(tibble)

https://tibble.tidyverse.org/

- or created from vectors (as for the `data frame`)

```
dct <- data.frame(exam1 = c(27,28,24,24,30,26,23,23,24,28,27,25),
                  exam2 = exam2, gender)
```

Tibbles vs. data.frame : printing

- two main differences in the usage of a tibble versus a data.frame:
- printing and subsetting
- Tibbles have a refined print method that shows only the first 10 rows:

```
atb <- tibble( a = lubridate::now() + runif(1e3) * 86400,
               b = 1:1e3,
               c = runif(1e3),
               d = sample(letters, 1e3, replace = TRUE) )
atb
%> # A tibble: 1,000 x 4
%>    a                       b      c d
%>    <dttm>              <int>  <dbl> <chr>
%>  1 2020-03-24 08:56:26     1 1.00   q
%>  2 2020-03-24 10:57:29     2 0.996  z
%>  3 2020-03-24 00:32:33     3 0.620  r
%>  4 2020-03-24 01:16:23     4 0.804  d
%>  5 2020-03-24 03:32:17     5 0.311  e
%>  6 2020-03-24 00:22:27     6 0.206  u
%>  7 2020-03-23 12:58:48     7 0.0390 e
%>  8 2020-03-23 20:36:03     8 0.449  d
%>  9 2020-03-24 01:29:00     9 0.271  r
%> 10 2020-03-24 10:52:01    10 0.460  v
%> # with 990 more rows
```


Tibbles vs. data.frame : subsetting

- tibbles can extract by name or position

```
tb1 <- tibble( x = runif(5),
               y = rnorm(5) )

# Extract by name
tb1$x
#> [1] 0.7330 0.2344 0.6604 0.0329 0.4605
tb1[["x"]]
#> [1] 0.7330 0.2344 0.6604 0.0329 0.4605

# Extract by position
tb1[[1]]
#> [1] 0.7330 0.2344 0.6604 0.0329 0.4605
```

- `tibbles` are more strict than `data.frame`: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist

```
tdc <- as_tibble(dc)
```

```
dc$gender
#>  [1] M F M ... F M F
#> Levels: F M

dc$gen
#>  [1] M F M ... F M F
#> Levels: F M

.
```

```
tdc$gender
#>  [1] M F M ... F M F
#> Levels: F M

tdc$gen
#> NULL
#> Warning message:
#> Unknown or uninitialised column:
  'gen'.
```

Data Input

Data Input

- numbers can be `input`ed through the keyboard, from the Clipboard, from an external file on disk, or from an external file on the Web
- use the concatenate function for up to 10 numbers
- and `scan()` for typing or pasting data into a vector

```
y <- c (6,7,3,4,8,5,6,2)

tu <- scan()
%> 1: 6
%> 2: 3
%> 3: 4
%> 4: 2
%> 5:
%> Read 4 items
tu
%> [1] 6 3 4 2
```

- but the easiest way is to read data from a file (or from the Web), already shaped in a data frame format

Data Input using `read.table()`

- the `read.table()` function reads data from a local file and creates a `data frame`

```
data <- read.table("yield.txt",header=T)

data
%>    year wheat barley oats rye corn
%> 1  1980   5.9    4.4  4.1 3.8  4.4
%> 2  1981   5.8    4.4  4.3 3.7  4.1
%> 3  1982   6.2    4.9  4.4 4.1  4.0
...
%> 27 2006   8.0    5.9  6.0 6.1  4.5
%> 28 2007   7.2    5.7  5.5 5.7  3.9
%> 29 2008   8.3    6.0  5.8 6.1  4.4
```

- the parameter `header = T` tells R to use the first row as column names

```
names(data)
%> [1] "year"   "wheat"  "barley" "oats"   "rye"    "corn"

str(data)
%> 'data.frame':        29 obs. of  6 variables:
%>  $ year  : int  1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 ...
%>  $ wheat : num  5.9 5.8 6.2 6.4 7.7 6.3 7 6 6.2 6.7 ...
%>  $ barley: num  4.4 4.4 4.9 4.7 5.6 5 5.2 5 4.7 4.9 ...
%>  $ oats  : num  4.1 4.3 4.4 4.3 4.9 4.6 5.2 4.6 4.6 4.5 ...
%>  $ rye   : num  3.8 3.7 4.1 3.7 4.7 4.6 4.7 4.8 4.6 4.8 ...
%>  $ corn  : num  4.4 4.1 4 4.1 4.7 4.3 4.3 4.5 4.2 3.8 ...
```


Data Input using `read.table()`

- if the separator between variable names and data fields are `not blanks` or `tabs`, (`\t`), a different separator can be specified with the `sep=","` option

```
datav <- read.table("bowens.csv",sep=",", header=T)

str(datav)
%> 'data.frame':        733 obs. of  3 variables:
%>  $ place: Factor w/ 727 levels "Abingdon","Admoor␣Copse",..: 1 2 3 ..
%>  $ east : int  50 60 48 70 59 60 60 59 61 60 ...
%>  $ north: int  97 70 87 73 65 65 63 66 63 67 ...
```

read.table() : separators and decimal points

- the default field separator character in read.table() is sep=" ": which identifies with one or more spaces, one or more tabs (\t), and one or more newlines (\n)
- for comma-separated fields use read.csv()
- for semicolon-separated fields use read.csv2()
- for tab-delimited fields with decimal points as a commas, use read.delim2()

```
File: bowens.csv
--------------------
|place,east,north  |
|Abingdon,50,97    |
|Admoor Copse,60,70|
|...               |
|Youlbury,48,3     |
--------------------
```

```
str(bw)
%> 'data.frame':   733 obs. of  3 variables:
%>  $ place: Factor w/ 727 levels "AERE␣Harwell",..: 2 3 1 4 5 ...
%>  $ east : int  50 60 48 70 59 60 60 59 61 60 ...
%>  $ north: int  97 70 87 73 65 65 63 66 63 67 ...
```


read.csv() and read.delim()

- additional functions to read a file in table format exist

```
> ?read.table
...
read.delim(file, header = TRUE, sep = "\t", quote = "\"",
␣␣␣␣␣␣␣␣␣␣dec␣=␣".",␣fill␣=␣TRUE,␣comment.char␣=␣"",␣...)
...
read.csv(file,␣header␣=␣TRUE,␣sep␣=␣",",␣quote␣=␣"\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
...
read.csv2(file, header = TRUE, sep = ";", quote = "\"",
␣␣␣␣␣␣␣␣␣␣dec␣=␣",",␣fill␣=␣TRUE,␣comment.char␣=␣"",␣...)
...
read.delim2(file,␣header␣=␣TRUE,␣sep␣=␣"\t",␣quote␣=␣"\"",
            dec = ",", fill = TRUE, comment.char = "", ...)
```

- further detailed instructions in the 'R Data Import/Export' manual:
  https://cran.r-project.org/doc/manuals/r-release/R-data.html

Data Input from the Web and from DB

- R can read data form the network using HTTP by specifying the file URL

```
wc <- read.table("https://tinyurl.com/murders-txt", header=T)

str(wc)
%> 'data.frame':    50 obs. of  4 variables:
%>  $ state      : Factor w/ 50 levels "Alabama","Alaska",..: 1 2 ...
%>  $ population: int  3615 365 2212 2110 21198 2541 3100 ...
%>  $ murder     : num  15.1 11.3 7.8 10.1 10.3 6.8 3.1 6.2 ...
%>  $ region     : Factor w/ 4 levels "North.Central",..: 3 4 4 ...
```

- several packages available on CRAN to help R communicate with DBMSs:

  combining a unified 'front-end' package with a 'back-end' module, several common relational databases can be accessed (RMySQL, ROracle, RPostgreSQL and RSQLite)

- finally, R can read binary data files: NASA's HDF5 (Hierarchical Data Format, https://www.hdfgroup.org/HDF5/) and UCAR's netCDF data files (network Common Data Form, http://www.unidata.ucar.edu/software/netcdf/)

- and image files

Example: data Input from the Web

- let's retrieve the latest data on the COVID-19 Virus infection from the European Centers for Disease Control https://www.ecdc.europa.eu/en

- R can read data form the network using HTTP by specifying the file URL

Example: data Input from the Web

- we download an EXCEL file
- we use the following packages: `lubridate`, `curl` and `readxl`

```
url <-  "https://www.ecdc.europa.eu/sites/default/files/documents/"
fname <- "COVID-19-geographic-disbtribution-worldwide-"
date <- lubridate::today() - 1
ext = ".xlsx"
target <- paste(url, fname, date, ext, sep="")
message("target: ", target)

tmp_file <- tempfile("data", "/tmp", fileext=ext)
tmp <- curl::curl_download(target, destfile=tmp_file)
```

- data are imported in a `tibble` data structure

```
(data <- readxl::read_xlsx(tmp_file))
%> A tibble: 6,012 x 8
%>   DateRep             Day Month  Year Cases Deaths Countries. GeoId
%>   <dttm>            <dbl> <dbl> <dbl> <dbl>  <dbl> <chr>      <chr>
%> 1 2020-03-21 00:00:00  21     3  2020     2      0 Afghanistan   AF
%> 2 2020-03-20 00:00:00  20     3  2020     0      0 Afghanistan   AF
%> 3 2020-03-19 00:00:00  19     3  2020     0      0 Afghanistan   AF
%> 4 2020-03-18 00:00:00  18     3  2020     1      0 Afghanistan   AF
%> 5 2020-03-17 00:00:00  17     3  2020     5      0 Afghanistan   AF
%> 6 2020-03-16 00:00:00  16     3  2020     6      0 Afghanistan   AF
%> 7 2020-03-15 00:00:00  15     3  2020     3      0 Afghanistan   AF
%> 8 2020-03-11 00:00:00  11     3  2020     3      0 Afghanistan   AF
%> 9 2020-03-08 00:00:00   8     3  2020     3      0 Afghanistan   AF
%>10 2020-03-02 00:00:00   2     3  2020     0      0 Afghanistan   AF
% ... with 6,002 more rows
```


Homework

- learn how to extract and manipulate data from the imported `tibble`
- in the next lecture we will see how to represent data with histograms and plots

R graphics

Alberto Garfagnini

Università di Padova

R lecture 5

Saving in R (1)

Saving R objects

- sometimes we need to save object created in R
- to save the current R session, so that it can be loaded at a later stage to continue working on it:

```
save(list = ls(all=TRUE), file = "my-session")
```

- a binary file will be produced and saved on disk
- everything can be loaded, at a later stage, with the following command:

```
load(file= "my-session")
```

Saving R history

- sometimes we need to save only the lines of code that have been typed in an R session

```
savehistory(file = "my-history.R")
```

- a text file with all the command is saved on disk
- to retrieve history, type:

```
loadhistory(file = "my-history.R")
```

Saving in R (2)

Saving graphics

- graphics can be saved in either pdf or postscript to include them in a report
- the procedure is to open a new pdf or postscript device, with the `pdf()` or `postscript()` functions
- then all commands needed to create the graphics can be typed in the R session, and once finished, the device has to be closed with the `dev.off()` function. Example:

```
pdf("my-plot.pdf")
hist(rnorm(10000))
dev.off()
```

Saving data produced within R

- let's suppose we have produced a vector we want to save on disk

```
nbnumbers <- rnbinom(1000, size=1, mu=1.2)
```

- and we want to save them in a file, in a single column

```
write(nbnumbers,"nbnumbers.txt",1)
```

- if, instead, we want to save them in a matrix like format

```
xmat <- matrix(rpois(100000,0.75),nrow=1000)
write.table(xmat,"table.txt",col.names=F,row.names=F)
```

- we have saved 1000 rows each of 100 Poisson random numbers with $\lambda = 0.75$

R graphics systems

base graphics

- a pen on paper model : you can only draw on top of a plot, no modification or deletion of existing content possible
- no user accessible representation of a graphics, only appearance on the screen
- fast primitives, but with limited scope

grid graphics

- developed by Paul Murrell (started in his PhD work)
- graphical objects can be represented independently of the plot and modified later
- a system of viewports makes it easier to lay out complex graphics

lattice graphics

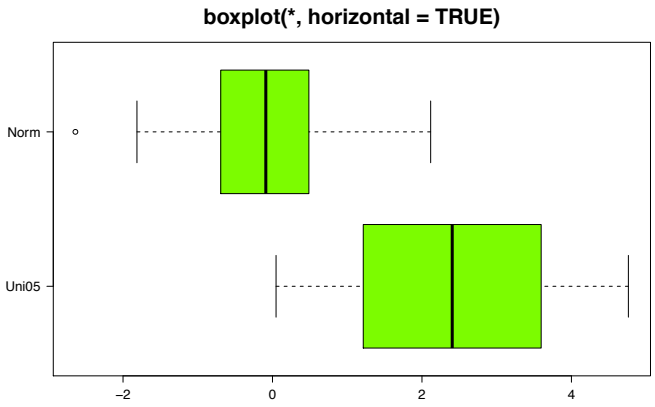
- developed by Deepayan Sarkar
- use grid graphics to implement the trellis graphics system of Cleveland

ggplot2

- a data visualization package creatd by Hadley Wickham in 2005
- it implements L. Wilkinson's Grammar of Graphics: a general scheme for data visualization which breaks up graphs into semantic components such as scales and layers

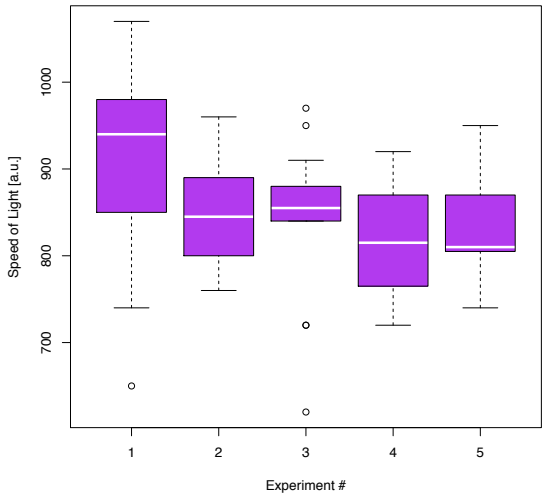
The R `boxplot()` function

- is a one-dimensional plot, known also as the box-and-whisker plot
- may be displayed vertically or horizontally
- the boxplot is always based on three quantities: top and bottom of the box are determined by the upper and lower quantiles; the band inside the box is the median
- the whiskers are created according to the purpose of the analyses and defined by the experimenter




```
mat <- cbind(Uni05 = (1:100)/21,
             Norm = rnorm(100))
df1 <- as.data.frame(mat)
par(las = 1)
boxplot(df1, horizontal = TRUE)

boxplot(Speed ~ Expt,
        data = morley,
        xlab = "Experiment #",
        ylab = "Speed of Light",
         col = "darkorchid2",
      medcol = "white")
```



The R `hist()` function

- an `histogram` object has a complex structure
- its data can be accessed using the `$ + name` syntax; as example, `hm$breaks` is a vector with the bin limits

```
hm <- hist(morley[,3], col="darkolivegreen3",
           xlab="Speed of light [a.u.]", main="Michelson Morley")

str(hm)
#> $breaks
#>  [1]  600  650  700  750  800  850  900
#>  [8]  950 1000 1050 1100

#> $counts
#>  [1]  2  0  7 16 30 22 11 11  0  1

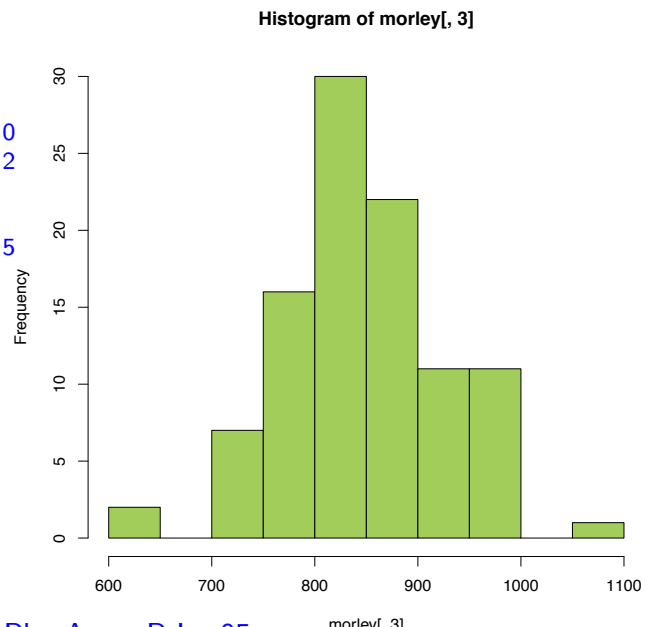
#> $density
#>  [1] 0.0004 0.0000 0.0014 0.0032 0.0060
#>  [6] 0.0044 0.0022 0.0022 0.0000 0.0002

#> $mids
#>  [1]  625  675  725  775  825  875  925
#>  [8]  975 1025 1075

#> $xname
#> [1] "morley[, 3]"

#> $equidist
#> [1] TRUE

#> attr(,"class")
#> [1] "histogram"
```



hist() function main parameters

- `hist(v, main, xlab, xlim, ylim, breaks, col, border)`
- `v` a vector with numeric values used in histogram
- `main` indicates title of the chart
- `col` is used to set color of the bars
- `border` is used to set border color of each bar
- `xlab` is used to give description of x-axis
- `xlim` specifies the range of values on the x-axis
- `ylim` stands for the range of values on the y-axis
- `breaks` is used to mention the width of each bar
- `frame = FALSE` removes the box around the plot

Superimposing histograms

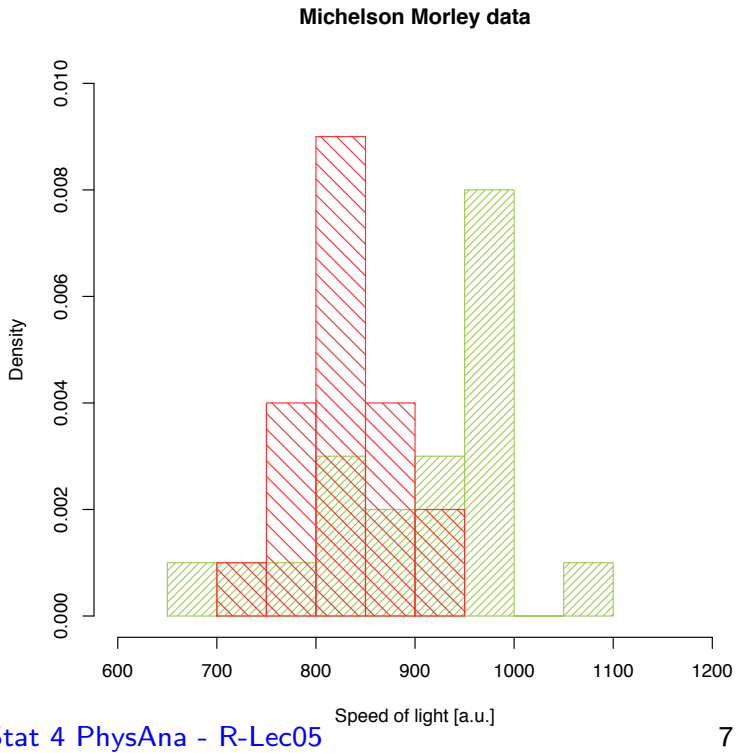
- the `add = TRUE` histogram parameters will do the job
- the option `freq = TRUE` will ensure that, in case of non equal number of observations, the heights of an interval remain the same

```
exp_1 <- morley[[3]][morley$Exp==1]
exp_5 <- morley[[3]][morley$Exp==5]

hist(exp_1, col="darkolivegreen3",
             density=20, freq=TRUE,
             xlim=c(600,1200),
             ylim=c(0,0.01),
     xlab="Speed of light [a.u.]",
     main="Michelson Morley data")

hist(exp_5, col="firebrick1",
             density=10, freq=TRUE,
             angle=-45,
             add=TRUE)

legend(x=1100, y=6.5,
       c("exp_1", "exp_5"),
       col=c("darkolivegreen3",
             "firebrick1"),
       pch="-", cex=1.25)
```



More plots on one page

- through `par()` it is possible to query or specify graphical parameters
- we divide the plot area in 2-row, 3-columns
- but since we have only 5 histograms, this leaves an empty plot area

  CheatSheet: https://raw.githubusercontent.com/rstudio/cheatsheets/master/how-big-is-your-graph.pdf

```
# Save the old par versions

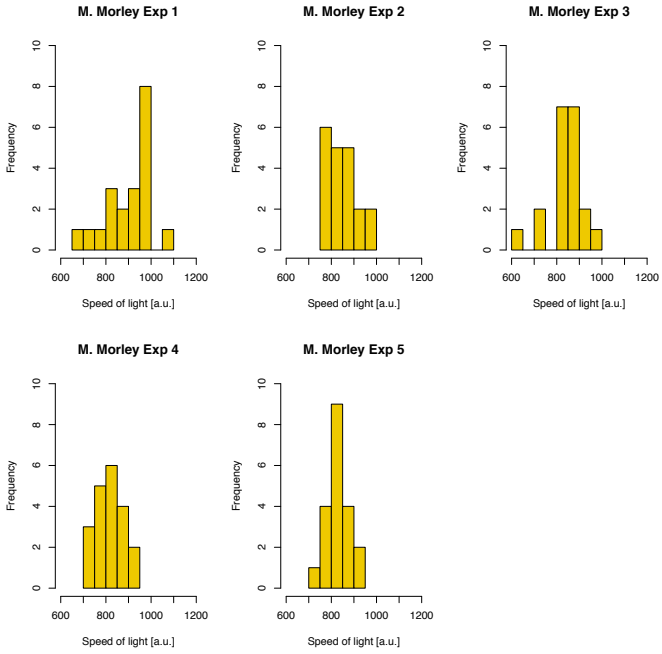
old_par <- par()

# Divide the graphical area in
#
2-rows, 3 columns

par(mfrow=c(2,3))

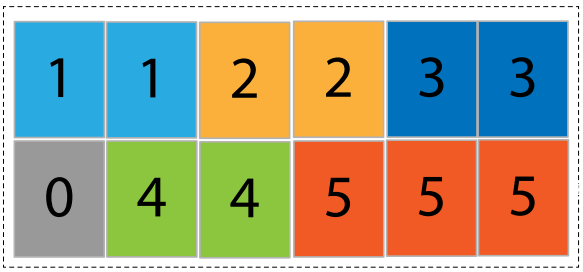
for (n_exp in 1:5) {
   h_text <- paste("M. Morley exp",
                   n_exp, sep=" ")

   hist(morley[morley$Exp==n_exp,3],
        col="gold2",
        xlim=c(600,1200),
        ylim=c(0,10),
        xlab="Speed of light [a.u.]",
        main=h_text)
}
```



More plots on the page with `layout()`

- the `layout(mat)` function divides the device up into as many rows and columns as there are in matrix 'mat'
- a value of 0 says that such parts should not be used for plots



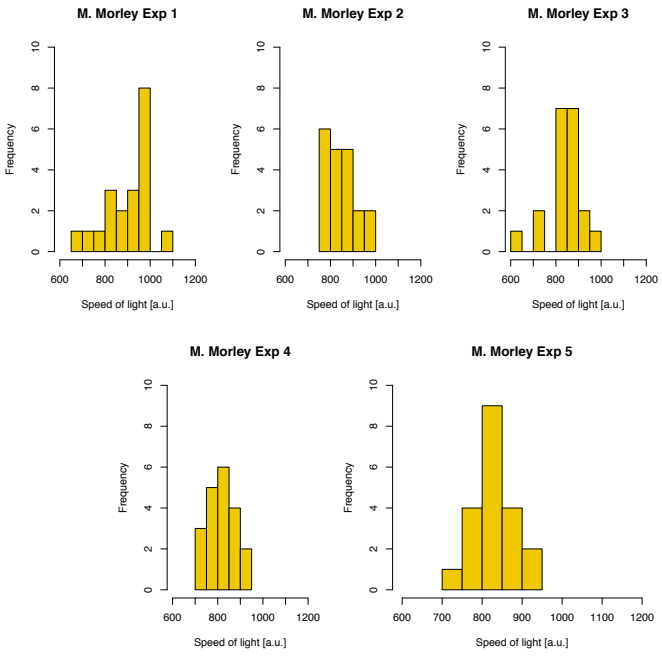

```
# Divide the area: 2 rows 6 columns

p_area <- matrix(c(1,1,2,2,3,3,
                   0,4,4,5,5,5),
                 nrow=2, ncol=6,
                 byrow=TRUE)


p_area
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    2    2    3    3
[2,]    0    4    4    5    5    5

layout(p_area)

for (n_exp in 1:5) {
   h_text <- paste("M. Morley exp",
                   n_exp, sep=" ")
   hist(morley[morley$Exp==n_exp,3],
        col="gold2",
        xlim=c(600,1200),
        ylim=c(0,10),
        xlab="Speed of light [a.u.]",
        main=h_text)
}
```



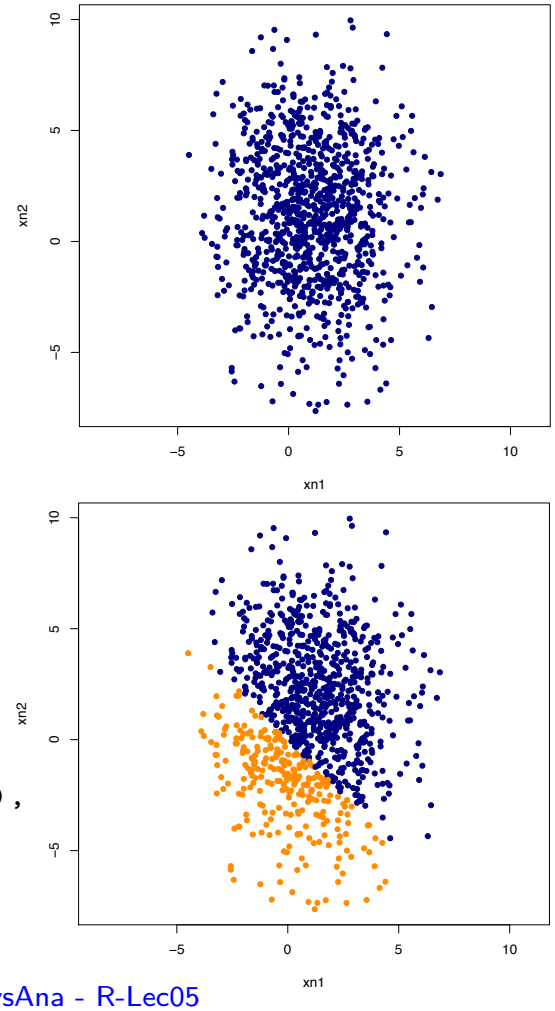
The R scatter `plot()` function

- the `plot()` function allows to produce a scatter plot of one variable versus the other
- the `asp = value` parameter allows to keep the y/x aspect ratio to a fixed value
- `asp=1` sets the same scale for both x and y axis, even if the plot window is re-scaled

```
set.seed(34761542)
xn1 <- rnorm(1000, 1, 2)
xn2 <- rnorm(1000, 1, 3)
plot( xn1, xn2, pch = 20,
      col = "navy", cex=1.25,
      asp = 1 )
```

- it is possible to use a third variable for a color

```
xcontrol <- jitter(xn1+xn2, 2)
plot( xn1, xn2, pch = 20, cex=1.25,
      col = ifelse(xcontrol>0,
                   "navy", "darkorange"),
     asp = 1 )
```

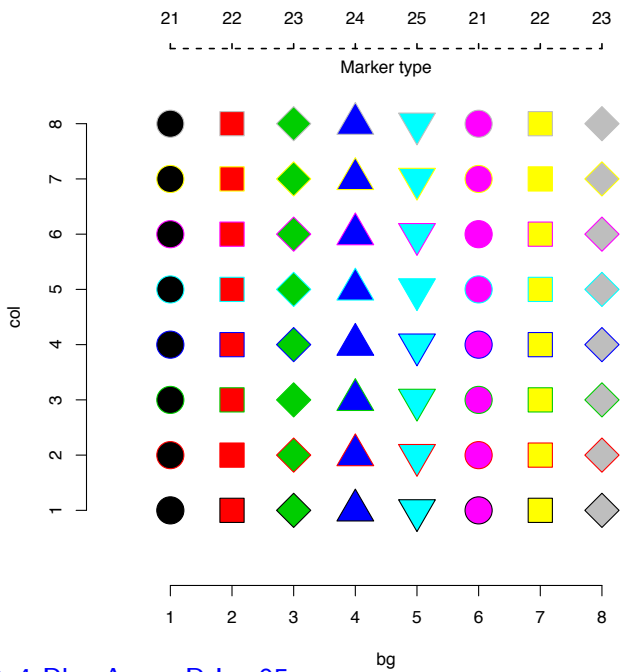


Colors for symbols and axes

- with the parameter `pch` it is possible to specify the plotting symbols
- moreover, it is possible to set the background `bg`, and fill Colors `col`, separately
- the option `freq = TRUE` will ensure that, in case of non equal number of observations, the heights of an interval remain the same

```
plot(0:9, 0:9,
     type="n", axes=FALSE,
     ylab="col", xlab="bg")
for (i in 1:8) {
     points(1:8, rep(i,8),
            pch=c(21,22,23,24,25),
            bg=1:8, col=i, cex=3.5)
}
axis(1, at=1:8)
axis(2, at=1:8)

axis(3, at=1:8,
     c(21,22,23,24,25,21,22,23),
     lty=2, lwd=1.5)
text(4.5, 9, "Marker type")
```

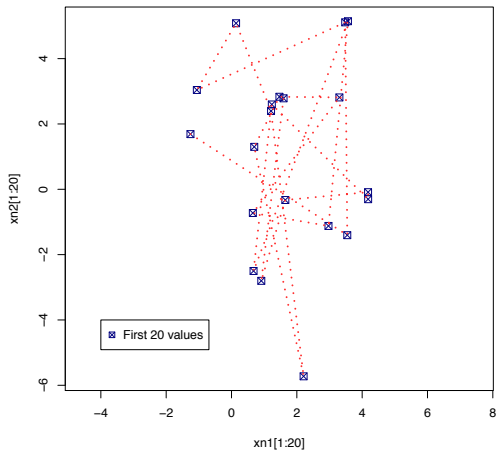


Joining points with `lines()`

- the primitive `lines()` allows to connect points with lines
- the line go across the points

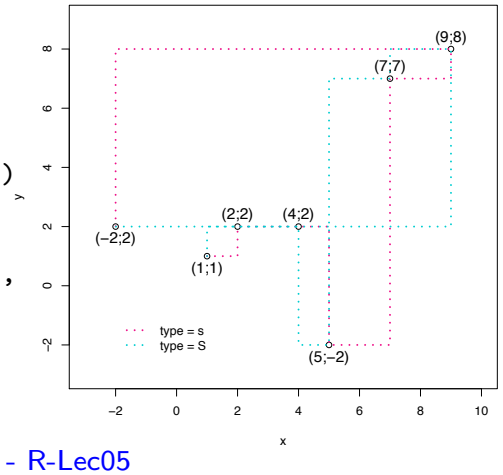
```
plot( xn1[1:20], xn2[1:20],
      pch = 7, cex=1.25, col = "navy")

lines(xn1[1:20], xn2[1:20],
      col = "firebrick1", lty=3)
```



- with option `type='s'`, a stepped line going across first and then up (or down)
- with option `type='S'`, a stepped line goes first up (or down) and then across

```
lines(x, y, col="deeppink2", type="s")
lines(x, y, col="darkturquoise", type="S")
tpos <- c(1,3,3,1,3,3,1)
text(x, y,
     labels=paste("(",x,";",y,")",sep=""),
     pos=tpos, offset=0.5, cex=1.25)
```



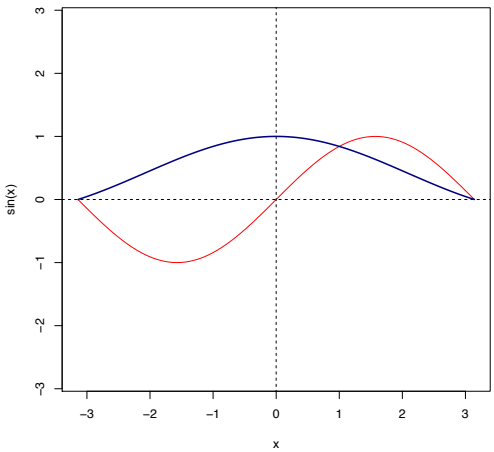
Plotting curves

- `curve()` allows to plot an analytical function
- `abline()` draws a line, given the intercept and slope

```
curve(sin(x), -pi, pi, col="red", asp=1)

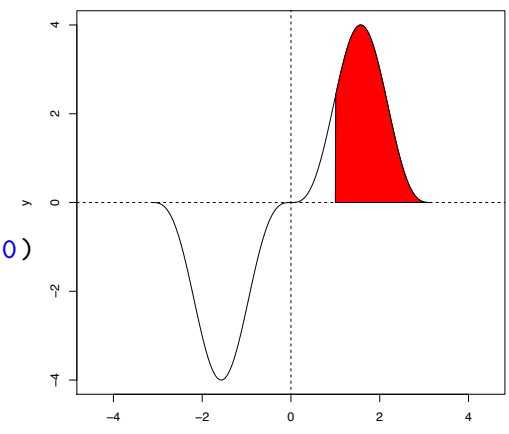
# The x-axis, going through (0,0)
abline(0,0,lty=2)
# The y-axis, going through (0,0)
abline(0,10000,lty=2)

curve(sin(x)/x, -pi, pi, col="navy",
      lw=2, add=T) # add to plot
```




```
# Similar way to plot a curve
x <- seq(-pi, pi, 0.01)
y <- 4*sin(x)^3
plot(x,y, type="l", asp=1)

polygon( c(1,x[x>=1]),
         c(0,y[x>1]), col="red", angle=10)
abline(0,0, lty=2)
abline(0,10000, lty=2)
```

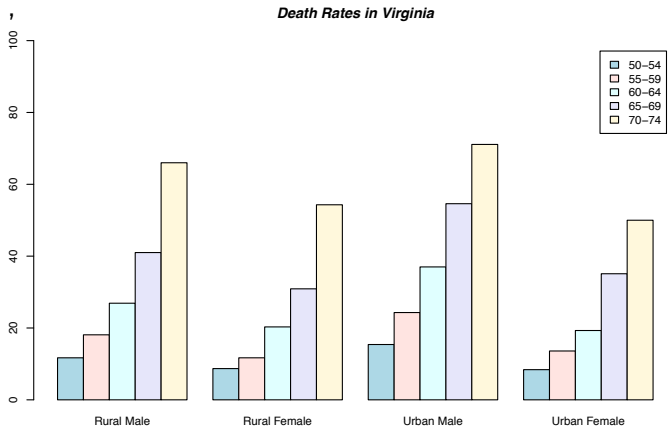


barplot

- a barplot shows the relationship between a numeric variable and a categorical variable
- R creates a barplot with vertical or horizontal bars
- be careful: a barplot is not an histogram !

```
barplot(VADeaths, beside = TRUE,
        col = c("lightblue",
                "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths),
          ylim = c(0, 100))
title(main = "Death Rates in Virginia")
```

```
VADeaths
#>       Rural   Rural    Urban   Urban
#>        Male Female     Male Female
#> 50–54  11.7    8.7     15.4    8.4
#> 55–59  18.1   11.7     24.3   13.6
#> 60–64  26.9   20.3     37.0   19.3
#> 65–69  41.0   30.9     54.6   35.1
#> 70–74  66.0   54.3     71.1   50.0
```



The Grammar of Graphics

- created by Wilkinson in 2005 to describe the features living behind all statistical graphics
- it has the following components:

- layer: are used to create the objects on a plot. They are defined by five basic parts: data (the source of the information to be visualized), mapping (how variables are applied to the plot), statistical transformation (which transform the data by summarizing the information), geometric object (controls the type of the plot to be created) and position adjustment
- scale : controls how data is mapped to aesthetic attributes (ex: scale for colors)
- coordinate system : maps the position of objects onto the plane of the plot and controls how axes and grid lines are drawn
- faceting : can be used to split data into subsets of the entire dataset

Wilkinson L., *The grammar of graphics. Statistics and computing* 2005, Springer, New York

ggplot2 example

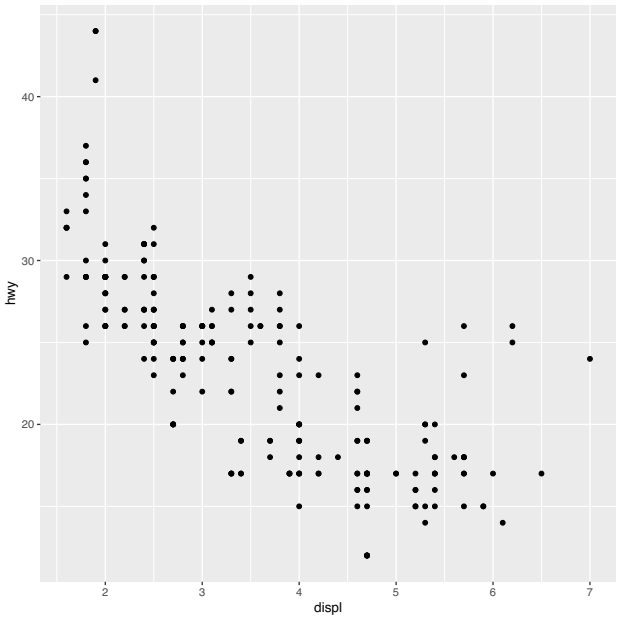
- every plot has three components :

1▷ data to be plotted

2▷ aesthetics, a set of mappings between variables in the data and visual properties

3▷ geoms, a layer that describes how to render each observation

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



- the produced scatter plot is defined by:

1▷ data = mpg dataframe

2▷ aesthetics = the x position is the engine mapping, while y is the fuel economy

3▷ geom = points

Important : data and aesthetic mappings are supplied in ggplot(), additional layers are added on with the '+' operator

ggplot2 example

- once a plot is created, it is possible to draw it using different rendering

```
p1 <- ggplot(mpg, aes(displ, hwy))

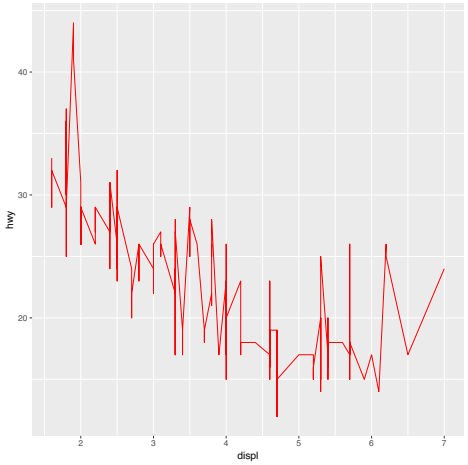

# the plot of the previous example
p1 + geom_point()

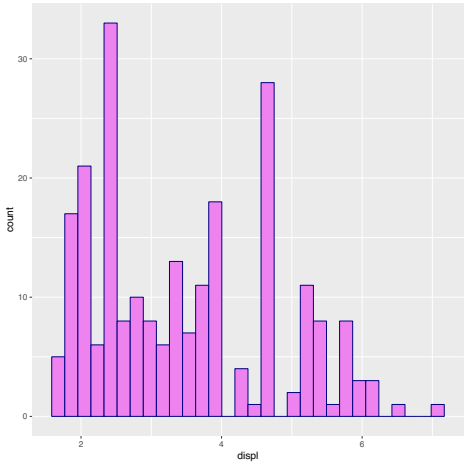

# new : points connected with lines
p1 + geom_line()


p2 <- ggplot(mpg, aes(displ))

p2 + geom_histogram(col="navy",
                    fill="violet")

p2 + geom_histogram(col='navy',
                    fill='orchid2',
                    binwidth=0.3)
```





ggplot2 barchart example

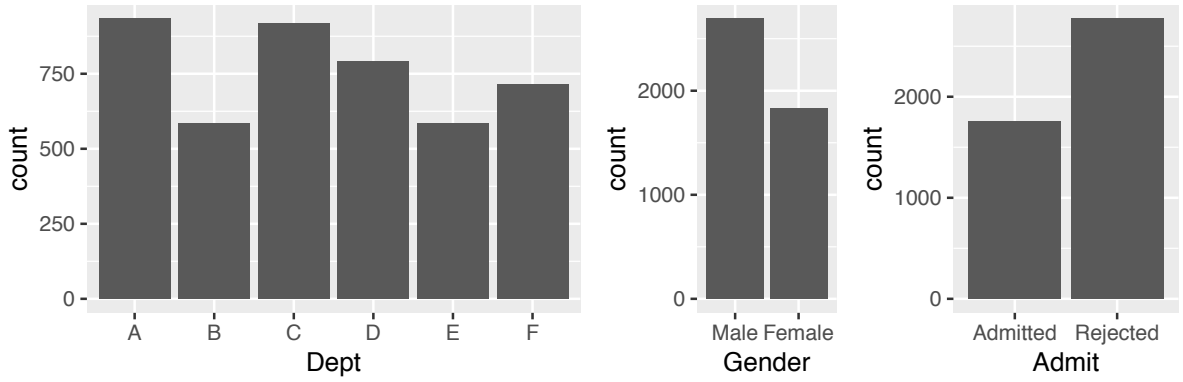
- the UCBAdmissions data set contains data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex
- we create separate barcharts for the variables: department, gender, and admitted or rejected

```
library(gridExtra)

ucba <- as.data.frame(UCBAdmissions)

a <- ggplot(ucba, aes(Dept)) + geom_bar(aes(weight=Freq))
b <- ggplot(ucba, aes(Gender)) + geom_bar(aes(weight=Freq))
c <- ggplot(ucba, aes(Admit)) + geom_bar(aes(weight=Freq))

grid.arrange(a, b, c, nrow=1, widths=c(7,3,3))
```



References

Books

- P. Murrell, *R graphics*, Chapman & Hall/CRC, 2006, 978-1-58488-486-6
- D. Sarkar, *Lattice: Multivariate Data Visualization with R*, Use R! series, Springer, 2008, 978-0-387-75968-5
  http://lmdvr.r-forge.r-project.org/figures/figures.html
- H. Wickham, *ggplot2, Elegant Graphics for Data Analysis*, Use R! series, Springer, 2016, 978-3-319-24275-0
  https://ggplot2-book.org/

Tutorials

- P. Murrell, *Introduction to R graphics*,
  https://www.stat.auckland.ac.nz/~paul/RGraphics/chapter1.pdf
  https://www.stat.auckland.ac.nz/~paul/RGraphics/RGraphicsChapters-1-4-5.pdf
- https://www.cedricscherer.com/2019/08/05/a-ggplot2-tutorial-for-beautiful-plotting-in-r/

Dates and time in R

R date/time

- measurement of time is highly idiosyncratic. Successive years start on different days of the week. There are months with different number of days. Leap years help complicating, adding an extra day on February every fourth year
- notations is also different: European, Asiatic and Americans put the day and the month in different years: 3/4/2006 can be the 3^{rd} of April or the 4^{th} of March.
- the `Sys.time()` function shows how dates and times is handled in R

```
#> Sys.time()
#> [1] "2021-03-23 21:00:57 CET"
```

- The baseline for expressing today's date and time in seconds is January 1^{st}, 1970:

```
(tnow <- Sys.time())
#> [1] "2021-03-23 21:02:31 CET"
as.numeric(tnow)
#> [1] 1616529752
```

- R uses `POSIX` system for representing dates and times

```
class(Sys.time())
#> [1] "POSIXct" "POSIXt"
```

R date/time classes : POSIXlt and POSIXct

- POSIXlt gives a list containing separate vectors for the year, month, day of the week, day within the year, ...
- it is very useful as a categorical explanatory variable
- POSIXct gives a vector containing the date and time expressed as a continuous variable that you can use in regression models (it is the number of seconds since the beginning of 1970).
- it is possible to convert from one representation to the other

```
tnow <- Sys.time()
time.list <- as.POSIXlt(tnow)

class(as.POSIXlt(tnow))
#> [1] "POSIXlt" "POSIXt"

unlist(time.list)
#>        sec        min       hour       mday
#> "31.6148"        "2"       "21"       "23"
#>        mon       year       wday       yday
#>        "2"      "121"        "2"       "81"
#>      isdst       zone     gmtoff
#>        "0"      "CET"     "3600"
```


Reading date/times data from files

- once dates in the format DD/MM/YYYY are read with read.data, they are read as characters, by default

```
data <- read.table("dates.txt", header=TRUE)
head(data)
#>   cnt       date
#> 1   3 12/02/2021
#> 2   4 13/02/2021
#> 3  12 14/02/2021
#> 4   8 15/02/2021

attach(data)

date
#> [1] 12/02/2021 13/02/2021 14/02/2021 15/02/2021

class(date)
#> [1] "character"
```

- data are not recognized by R as being dates
- to convert a factor, or a character string into a POSIXlt object, the strptime() function is used

```
Rdate <-strptime(as.character(date), "%d/%m/%Y")
class(Rdate)
#> [1] "POSIXlt" "POSIXt"
```

strptime() function

```
str(unclass(Rdate))
#> List of 11
#>  $ sec    : num [1:4] 0 0 0 0
#>  $ min    : int [1:4] 0 0 0 0
#>  $ hour   : int [1:4] 0 0 0 0
#>  $ mday   : int [1:4] 12 13 14 15
#>  $ mon    : int [1:4] 1 1 1 1
#>  $ year   : int [1:4] 120 120 120 120
#>  $ wday   : int [1:4] 2 3 4 5
#>  $ yday   : int [1:4] 42 43 44 45
#>  $ isdst  : int [1:4] 0 0 0 0
#>  $ zone   : chr [1:4] "CET" "CET" "CET" "CET"
#>  $ gmtoff: int [1:4] NA NA NA NA
```

- let's add the R-formatted date to our data frame

```
data <- data.frame(data,Rdate)
data
#>   cnt       date      Rdate
#> 1   3 12/02/2021 2021-02-12
#> 2   4 13/02/2021 2021-02-13
#> 3  12 14/02/2021 2021-02-14
#> 4   8 15/02/2021 2021-02-15
```


Dates and times arithmetic's

The following calculations are possible:

- time $\pm$ number
- time1 - time2
- time1 `logical-op` time2
- where `logical-op` is one of `==`, `!=`, $<$, $<=$, $>$, $>=$

```
y2 <- as.POSIXlt("2021-02-18")
y1 <- as.POSIXlt("2021-01-26")

y2 - y1
#> Time difference of 23 days

y1 + y2
#> Error in `+.POSIXt`(y1, y2) :
#> binary '+' is not defined for "POSIXt" objects
```

The following calculations are possible:

- it is possible to add or subtract a number of seconds or a `difftime` object from a dat-time object, but they cannot be added
- always convert dates and times into `POSIXlt` objects before staring any calculations

The `difftime()` and as.difftime() functions

- evaluating the difference between two dates and times involves the `difftime()` function

```
difftime("2021-02-18","2021-01-26")
#> Time difference of 23 days
```

- if only the number of days is needed, use

```
as.numeric(difftime("2021-02-18","2021-01-26"))
#> [1] 23
```

- the same operation can be applied to times, as well

```
t1 <- as.difftime("12:43:12")
t2 <- as.difftime("7:00:00")
t1-t2
#> Time difference of 5.72 hours

as.numeric(t1-t2)
#> [1] 5.72
```


Generating sequences of dates

- it may be useful to generate sequences of dates by years, months, weeks or days

```
seq(as.POSIXlt("2019-08-01"), as.POSIXlt("2019-10-12"), "1␣week")
#> [1] "2019-08-01 CEST" "2019-08-08 CEST" "2019-08-15 CEST"
#> ...
#> [9] "2019-09-26 CEST" "2019-10-03 CEST" "2019-10-10 CEST"

seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2029-10-12"), "3␣years")
#> [1] "2019-04-01 CEST" "2022-04-01 CEST" "2025-04-01 CEST"
#> [3] "2028-04-01 CEST"
```

- a number, instead of a recognized character string, will be interpreted as a number of seconds

```
seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2019-04-12"), 1024)
#> [1] "2019-04-01 00:00:00 CEST" "2019-04-01 00:17:04 CEST"
#> ...
#> [929] "2019-04-11 23:57:52 CEST"
```

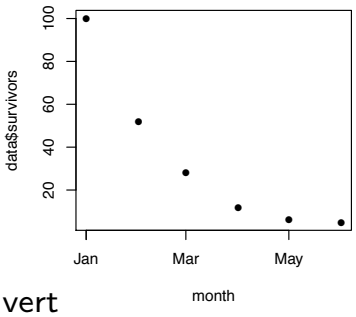
- as for other type of seq, the length of the vector can be specified instead of the final date:

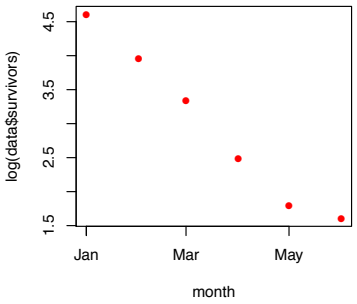
```
seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2019-04-12"), length=3)
#> [1] "2019-04-01 00:00:00 CEST" "2019-04-06 12:00:00 CEST"
#> [3] "2019-04-12 00:00:00 CEST"
```

Regression using dates and times : 1

- an experiment was performed observing the number of insects over 6 months

```
data <- read.table("timereg.txt", header=T)
data
#>   survivors       date
#> 1       100 01/01/2011
#> 2        52 01/02/2011
#> 3        28 01/03/2011
#> 4        12 01/04/2011
#> 5         6 01/05/2011
#> 6         5 01/06/2011
```





- as before, we use `strptime()` to convert a data string into a date-time object

```
dl <-strptime(data$date, "%d/%m/%Y")

class(dl)
#> [1] "POSIXlt" "POSIXt"

mode(dl)
#> [1] "list"

# Let's plot the data
par(mfrow=c(2,2))
plot(dl, data$survivors, pch=16, xlab="month")
plot(dl, log(data$survivors), pch=16, col="red", xlab="month")
```

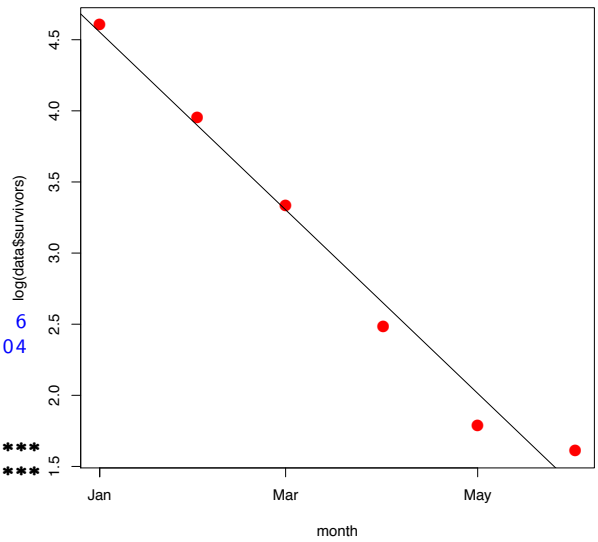

Regression using dates and times : 2

- plotting the data suggests an exponential decay in the survivor variable

```
model <- lm(log(data$survivors)~dl)
#> Error in model.frame.default(formula = log(data$survivors) ~ dl :
#>   invalid type (list) for variable 'dl'
```

- the reason for the error is that we cannot use a list as an explanatory variable in a linear model
- we need to convert from a list (`class = POSIXlt`) to a continuous numeric variable (`class = POSIXct`)

```
dc <- as.POSIXct(dl)
model <- lm(log(data$survivors)~dc)
abline(model)
summary(model)
```




```
Call:
lm(formula = log(data$survivors) ~ dc)

Residuals:
      1       2       3       4       5       6
 0.05178  0.05416  0.02792 -0.16395 -0.22195  0.25204

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.216e+02  2.286e+01   14.07 0.000148 ***
dc          -2.450e-07  1.758e-08  -13.94 0.000154 ***
```

The lubridate package https://lubridate.tidyverse.org/

- the packages makes it easier to work with date and times
- it allows to create date from strings:

```
ymd('2021-03-22')
#> [1] "2021-03-22"
d1 <- ymd('2021-03-22')
d2 <- mdy('03-22-2021')
d3 <- dmy('22-03-2021')

d1-d2; d1-d3
#> Time difference of 0 days
#> Time difference of 0 days
```

- but also from unquoted numbers

```
> dmy(22032021)
[1] "2021-03-22"
```

- it has simple functions to get and set components of a date-time, such as `year()`, `month()`, `day()`, `hour()`, `minute()` and `second()`

```
> day(dmy(22032021))
[1] 22
```

- it expands the mathematical operations to be performed with date-time objects: 3 new time span classes (`durations`, `periods` and `intervals`, borrowed from http://joda.org

The `tidyverse` collection of packages

Alberto Garfagnini

Università di Padova

R lecture 6

`tidyverse`

- it's an opinionated collection of R packages designed for data science.
- all packages share an underlying design philosophy, grammar, and data structures.
- Web Site: https://www.tidyverse.org/

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

tidyverse packages

ggplot2	gplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics
dplyr	it provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges
tidyr	it provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable
readr	it provides a fast and friendly way to read rectangular data (csv, tsv, and fwf)
purrr	it enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors
tibble	a modern re-imagining of the data frame
stringr	it provides a cohesive set of functions designed to make working with strings
forcats	it provides a suite of useful tools that solve common problems with factors

readr : https://readr.tidyverse.org/

- it provides a fast and friendly way to read rectangular data: csv, tsv, and fwf

- read_csv() : read and import comma separated (CSV) files
- read_tsv() : read and import tab separated (TSV) file
- read_delim() : read and import general delimited files
- read_fsw() : read and import fixed width files
- read_log() : read and import web log files

Alternatives

- in baseR : the read.table() function
- in data.table : the function fread() is similar to read_csv()

dplyr

https://dplyr.tidyverse.org/

- dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

function	description	SQL equivalent
select()	select on columns (i.e. variables)	SELECT
filter()	filter a subset of rows	WHERE
group_by()	group the data	GROUP BY
summarise()	reduces multiple values down to a single summary	-
arrange()	changes the ordering of the rows	ORDER BY
join()		JOIN
mutate()	adds new variables that are functions of existing variables	COLUMN ALIAS

- all function operate on a `data frame` and the result is a new `data frame`
- dplyr functions never modify their input

dplyr : filter()

- it allows to subset observations based on their values
- the first argument is the name of the data frame
- the second and subsequent arguments are the expressions that filter the data frame

```
filter(flights, month == 1, day == 1)
#> # A tibble: 842 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
                                                    sched_arr_time
#>   <int> <int> <int>   <int>          <int>     <dbl>    <int>
#> 1  2013     1     1     517            515         2      830
#> 2  2013     1     1     533            529         4      850
#> 3  2013     1     1     542            540         2      923
#> 4  2013     1     1     544            545        -1     1004
#> 5  2013     1     1     554            600        -6      812
#> 6  2013     1     1     554            558        -4      740
#> # with 836 more rows, and 11 more variables: ...
```

- `nycflights13::flights` is a data frame that contains all 336,776 flights that departed from New York City in 2013
- it's available in the `library(nycflights13)`

dplyr : arrange()

- it works similarly to `filter()` except that instead of selecting rows, it changes their order
- it takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
> arrange(flights, month,day, sched_dep_time)
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time
   <int> <int> <int>    <int>          <int>
 1  2013     1     1      517            515
 2  2013     1     1      533            529
 3  2013     1     1      542            540
 4  2013     1     1      544            545
```

- to rearrange a column in descending row, use `desc()`

```
> arrange(flights, month,day, desc(sched_dep_time))
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time
   <int> <int> <int>    <int>          <int>
 1  2013     1     1     2353           2359
 2  2013     1     1     2353           2359
 3  2013     1     1     2356           2359
 4  2013     1     1     2250           2255
```


dplyr : select()

- it allows to select a subrange of columns in the data frame

```
> select(flights, year, month, day, dep_time)
# A tibble: 336,776 x 4
    year month   day dep_time
   <int> <int> <int>    <int>
 1  2013     1     1      517
 2  2013     1     1      533
 3  2013     1     1      542
```

- usual selection rules apply: we can select a range of columns

```
> select(flights, dep_time:dep_delay)
# A tibble: 336,776 x 3
   dep_time sched_dep_time dep_delay
      <int>          <int>     <dbl>
 1      517            515         2
 2      533            529         4
 3      542            540         2
```

- we can remove columns with the - (minus) sign

```
> select(flights, -(year:day))
# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time
      <int>          <int>     <dbl>    <int>
 1      517            515         2      830
 2      533            529         4      850
 3      542            540         2      923
```

dplyr : mutate()

- besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns

```
flights_sml <- select(flights, year:day, ends_with("delay"),
                      distance, air_time)

> mutate(flights_sml, gain = dep_delay - arr_delay,
         speed = distance / air_time * 60)
# A tibble: 336,776 x 9
    year month  day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
 1  2013     1     1       2      11    1400     227    -9  370.
 2  2013     1     1       4      20    1416     227   -16  374.
 3  2013     1     1       2      33    1089     160   -31  408.
 4  2013     1     1      -1     -18    1576     183    17  517.
```

- if we want to keep only the new variables, we use `transmute()`:

```
transmute(flights, gain = dep_delay - arr_delay,
          hours = air_time / 60, gain_per_hour = gain / hours)
# A tibble: 336,776 x 3
    gain hours gain_per_hour
   <dbl> <dbl>         <dbl>
 1    -9 3.78          -2.38
 2   -16 3.78          -4.23
 3   -31 2.67         -11.6
 4    17 3.05           5.57
```


dplyr : summarise() and group_by()

- summarise() collapses a data frame to a single row
- it is very useful ewhen combined with group_by()
- group_by() takes an existing data frame and converts it into a grouped data frame where operations are performed by group

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>% group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
 A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.4
 2  2013     1     2 13.7
 3  2013     1     3 10.9
 4  2013     1     4  8.97
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.56
 9  2013     1     9  2.30
10  2013     1    10  2.84
```

The PIPE operator %>%

- it process a data-object with a sequence of operations by passing the result of one step as input for the next step using infix-operators rather than the more typical R method of nested function calls
- it is defined in the `magrittr` package, but it gained huge visibility and popularity with the `dplyr` package

Syntax

```
lhs %>% rhs # pipe syntax for rhs(lhs)

lhs %>% rhs(a = 1) # pipe syntax for rhs(lhs, a = 1)

lhs %>% rhs(a = 1, b = .) # pipe syntax for rhs(a = 1, b = lhs)

lhs %<>% rhs # pipe syntax for lhs <- rhs(lhs)

lhs %$% rhs(a) # pipe syntax for with(lhs, rhs(lhs$a))

lhs %T>% rhs # pipe syntax for { rhs(lhs); lhs }
```

- `lhs` = a value or the magrittr placeholder
- `hhs` = a function call using the magrittr semantics

The PIPE operator %>% - examples

Basic use

```
library(magrittr)

1:10 %>% mean
# [1] 5.5

# is equivalent to
mean(1:10)
# [1] 5.5

years <- factor(2008:2012)
as.numeric(as.character(years))

# piping equivalent
years %>% as.character %>% as.numeric
grepl("Wo", substring("Hello World", 7, 11))
#> [1] TRUE

"Hello World" %>% substring(7, 11) %>% grepl(pattern = "Wo")
#> [1] TRUE
"Hello World" %>% substring(7, 11) %>% grepl("Wo", .)
#> [1] TRUE
"Hello World" %>% substring(7, 11) %>% { c(paste(. ,'Hi', .)) }
#> [1] "World Hi World"
```

Combining multiple operation with the pipe

- the pipe, %>%, can be used to rewrite multiple operations in a compact way; it can be read left-to-right, top-to-bottom
- piping improves code readability

```
select(flights, year:day, ends_with("delay"),
       distance , air_time) %>%
       transmute(gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)
# A tibble: 336,776 x 2
    gain speed
   <dbl> <dbl>
 1    -9  370.
 2   -16  374.
 3   -31  408.
 4    17  517.
 5    19  394.
 6   -16  288.
 7   -24  404.
 8    11  259.
 9     5  405.
10   -10  319.
# ... with 336,766 more rows
```

- behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on

data.table

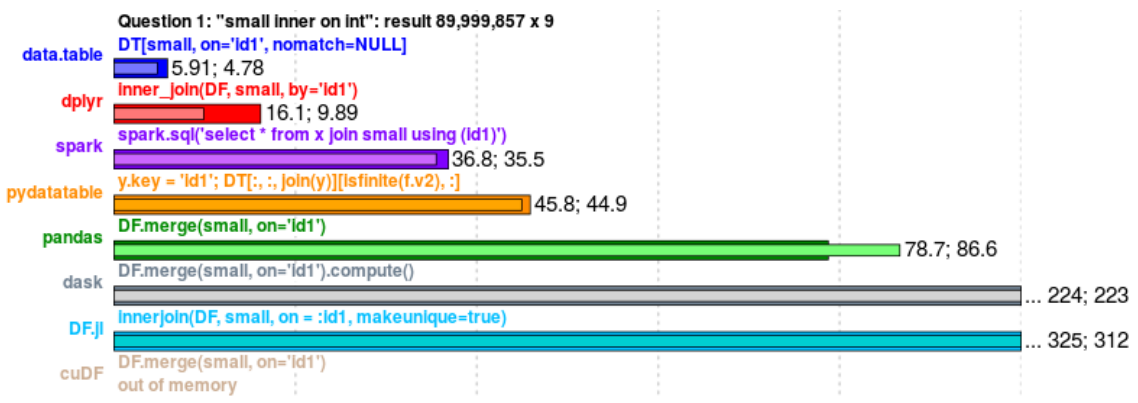
https://github.com/Rdatatable/data.table/wiki

for large dataframe

- it provides a high-performance version of base R's `data.frame`
- `data.table` is created using the `fread()` function for reading data on disk, or provided on the fly with the `data.table()` function

```
DT = data.table(
  id = c("b","a","a","c","c","b"),
  val = c(4,2,3,1,5,6)
)
```

- existing objects can be converted to `data.table` using the `setDT()` and the `as.data.table()` functions
- it is a optimized and runs faster for large data sets (example plot: 10^8 rows with 7 columns ➜5 GB data) https://h2oai.github.io/db-benchmark/



data.frame - 1

- have a 2D matrix like structure: rows and columns. We can:
  - subset rows (remove)

```
X[X$id != "a"]
```

  - select columns

```
X[, "val"]
```

  - and do it at the same time:

```
X[X$id != "a", "val"]
```

(remove rows and select column)

X

	id	val
1	b	4
2	a	2
3	a	3
4	c	1
5	c	5
6	b	6

data.frame - 2

- we can compute on columns:
  - sum column `valA` only for the rows where `code != "abd"`

```
sum(DF[DF$code != "abd", "valA"])
1.9
```

- we can perform operations on aggregated groups
  - sum `valA` and `valB` columns for `code != "abd"` and group by `id`

```
aggregate(cbind(valA, valB) ~ id,
          DF[DF$code != "abd", ], sum)
```

  - we can update values

```
DF[DF$code == "abd", "valA"] <- NA
```

DF

	id	code	valA	valB
1	1	abc	0.1	11
2	1	abc	0.6	7
3	1	abd	NA	5
4	2	apq	0.9	10
5	2	apq	0.3	13

	id	valA	valB
1	1	0.7	18
2	2	1.2	23

data.table

- they allow column names to be seen as variables within the [...]
- and computations can be done with them directly
- an additional argument, `by` is introduced
- a `data.table` has a row/column data structure, as `data.frames`
  - subset rows

```
X[id != "a", ]
```

  - select columns

```
X[, val]
```

  - and compute on columns

```
X[, mean(val)]
```

  - subset rows and select/compute on columns

```
X[X$id != "a", mean(val)]
```

  - and with a 'virtual 3rd dimension, group by

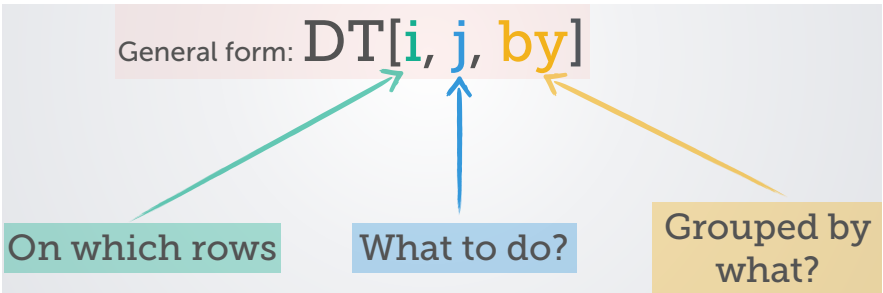
```
X[X$id != "a", .sum(valA), sum(valB), by=id]
```

X

	id	val
1:	b	4
2:	a	2
3:	a	3
4:	c	1
5:	c	5
6:	b	6

equivalence data.frame vs data.table

- think in terms of basic units: rows, columns and groups




```
sum(DF[DF$code != "abd", "valA"])
```

```
DT[code != "abd", sum(valA)]
```

```
aggregate(cbind(valA, valB) ~ id,
          DF[DF$code != "abd", ],
          sum)
```

```
DT[code != "abd",
   .(sum(valA), sum(valB)),
   by = id]
```

```
DF[DF$code == "abd", "valA"] <- NA
```

```
DT[code == "abd", valA := NA]
```

R functional programming

Alberto Garfagnini

Università di Padova

Advanced R 02

Function fundamentals

- R functions can be broken into 3 components:
  - arguments : the list of arguments that describe how to call the function
  - body : the code inside the function
  - environment : the data structure that tell us how the fucntion finds the values associated with the name

assign or it's a lambda function

```
mysum <- function(x, y) {
  # Compute the sum of 2 vectors
  x + y
}
```

```
> formals(mysum)
#> $x
#> $y
```

arguments of the function

```
body(mysum)
#> {
#>     x + y
#> }
```

inside the code of the function

```
environment(mysum)
#> <environment: R_GlobalEnv>
```

- functions, as objects, can have attributes

```
attributes(mysum)
#> $srcref
#> function(x, y) {
#>   # Compute the sum of 2 vectors
#>   x + y
#> }
```

```
attr(mysum, "srcref")
#> function(x, y) {
#>   # Compute the sum of 2 vectors
#>   x + y
#> }
```

Primitive functions

- are those found in the base package
- are primarily written in C, so their `formals()`, `body` and `environment()` are all `NULL`

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")

formals(sum)
#> NULL

body(sum)
#> NULL

environment(sum)
#> NULL

typeof(sum)
#> [1] "builtin"
```


Creating functions

A "named" function

1) create a function object with `function`
2) bind it to a name with `<-`

```
mym <- function(x) {
  sin(1 / x ^ 2)
}
mym(1:4)
#> [1] 0.84147098 0.24740396 0.11088263 0.06245932
```

Anonymous functions

- it is done when a function name (i.e. binding) is not given

```
integrate(function(x) sin(x) ^ 2, 0, pi)
#> 1.570796 with absolute error < 1.7e-14
```

List of functions

- functions can be put in a list

```
lfuns <- list(
            half = function(x) x/2,
            double = function(x) x*2)
lfuns$half(10)
#> [1] 5
lfuns$double(10)
#> [1] 20
```

Exercise for giov 8- tomorrow

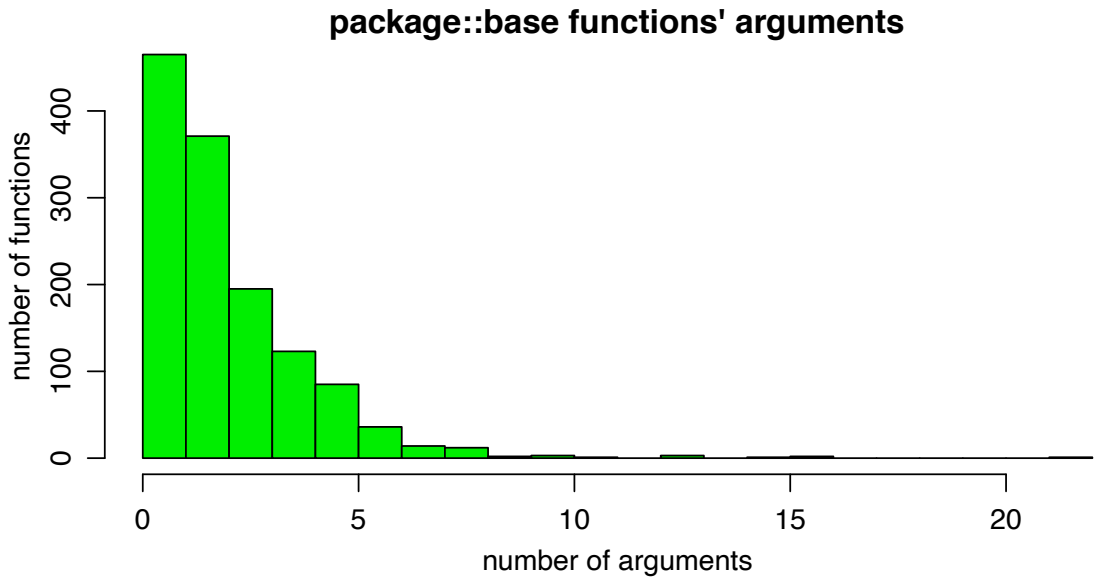
- the following code create a list of all functions in the base package

```
objs <- mget(ls("package:base", all=TRUE), inherits=TRUE)
bfuns <- Filter(is.function, objs)
```

↳ filter the list and apply a function

1➜ Determine the number of arguments for all functions and plot the distributions

2➜ How to restrict the search only to primitive functions ?



Functions calling

- R functions are normally invoked by placing the arguments in parentheses:

```
x <- c(1:3, NA, 5:10)
mean(x, na.rm=TRUE)
#> [1] 5.666667
```

- in case the functions arguments are inside a data structure
- the `do.call()` function can be called, instead:

```
x <- c(1:3, NA, 5:10)
args <- list(x, na.rm=TRUE)
do.call(mean, args)
#> [1] 5.666667
```

Functions composition

- let's imagine we need to call several functions:

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
x <- runif(10^3)
```

- we can nest the function calls

```
sqrt(mean(square(deviation(x))))
#> [1] 0.2925719
```

Functions calling (2)

- we could also store intermediate results as vectors

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
#> [1] 0.2925719
```

- but we could also use the `pipe` operator, `%>%`

```
library(magrittr)

x %>%
   deviation()  %>%
   square()  %>%
   mean()  %>%
   sqrt()
#> [1] 0.2925719
```

- `x %>% f()` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`

Lazy evaluation

Evaluated only in needed in R

- all function arguments are `lazy evaluated`

```
hstop <- function(x) { 10 }

hstop(1)
#> [1] 10

hstop(stop("This␣is␣an␣error!"))
#> [1] 10
```

stop the execution and gives us an error

```
stop("This␣is␣an␣error!")
#> Error: This is and error!
```

Promises

- unevaluated argument is called a promise, or a thunk.
- a promise is made up of two parts:
  - an expression, line `x + y` which gives rise to delayed computation
  - an environment, where the expression should be evaluated

Function arguments : default values

- function arguments can have default values

```
f <- function(a = 1, b = 2) c(a, b)
f()
#> [1] 1 2
```

default value

- since arguments are evaluated lazily, default arguments can be defined in terms of other arguments

```
g <- function(a = 1,  b = a * 2) c(a, b)
g()
#> [1] 1 2
g(10)
#> [1] 10 20
```

default value

lazy evaluated

evaluated

- if an argument was supplied or not can be seen with the `missing()` function

```
i <- function(a, b) { c(missing(a), missing(b)) }
i()
#> [1] TRUE TRUE
i(a=1)
#> [1] FALSE  TRUE
i(b=1)
#> [1]  TRUE FALSE
i(1,2)
#> [1] FALSE FALSE
```


The ... (dot-dot-dot) function argument

- it is a special argument called `...`
- it will match any arguments not otherwise matched, and can be easily passed on to other functions
- one relatively sophisticated user of `...` is the base `plot()` function
- `plot()` is a generic method with arguments `x`, `y` and `...`
- simple invocations of `plot()` end up calling `plot.default()` which has many more arguments (including `...`). In this way, plot() accepts graphical parameters which are listed in the help of par()

function that assign the color

```
plot(1:5, col = "red")
plot(1:5, cex = 5, pch = 20)

# The following allows to capture the arguments
f <- function(...) {
  names(list(...))
}
f(alpha=1, slope=3)
[1] "alpha" "slope"
```

Every operation is a function call

Golden rules

- everything that exists in R is an object
- but everything that happens is a function call
- this includes infix operators like +, control flow operators like for, if, and while, subsetting operators like [] and $, and even the curly brace {
- the backtick lets us refer to functions or variables that have otherwise reserved or illegal names

```
x <- 10; y <- 5;  x + y
[1] 15

`+`(x, y)
[1] 15

for (i in 1:2) print(i)
[1] 1
[1] 2

`for`(i, 1:2, print(i))
[1] 1
[1] 2

> { print(1)}
[1] 1
> `{`(print(1))
[1] 1
```


Every operation is a function call

- this allows to override the definitions of these special functions
- usually it is a bad idea, but it allows you to do something that would have otherwise been impossible
- example: we need to add 3 to every element of a list
- option 1: define a function add() and use sapply():

```
add <- function(x, y) x + y
sapply(1:10, add, 3)
[1] 4 5 6 7 8 9 10 11 12 13
```

- but we can also get the same effect using the built-in + function:

```
sapply(1:5, `+`, 3)
[1] 4 5 6 7 8

sapply(1:5, "+", 3)
[1] 4 5 6 7 8
```

- the second version works as well, because sapply() can be given the name of a function instead of the function itself
- it uses match.fun() to find functions given their names

Function arguments

- it is useful to distinguish between
- formal arguments ➜ a property of the function
- actual arguments ➜ can vary each time you call the function
- when calling a function, arguments can be specified by
- position, complete name, partial name
- arguments are matched first by exact name (perfect matching), then by prefix matching, and finally by position

```
f <- function(alpha, beta1, beta2) {
   list(a = alpha, b1 = beta1, b2 = beta2)
}
str(f(1,2,3))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(2,3,alpha=1))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(2,3,al=1))
List of 3
 $ a : num 1   $ b1: num 2   $ b2: num 3

str(f(1,2,beta=3))
Error in f(1, 2, beta = 3) : argument 3 matches multiple formal arguments
```


Special calls: Infix functions

- most functions in R are *prefix* operators: the name of the function comes before the arguments
- infix functions are those where the function name comes in between its arguments (for instance '+' or '-')
- all user created infix functions must start and end with %
- R comes with the following infix functions predefined: %%, %*%, %/%, %in%, %o%, %x%
- the complete list of built-in infix operators that don't need % is: ::, :::, $, @, ^, *, /, +, -, >, >=, <, <=, ==, !=, !, &, &&, |, ||, ~, <-, <<-
- we could create a new operator that pastes together strings:

```
`%+%` <- function(a, b) paste(a, b, sep = "")
"new" %+% "␣string"
[1] "new␣string"
```

- as far as R is concerned there is no difference between these two expressions:

```
"new" %+% "␣string"
[1] "new␣string"
`%+%`("new", "␣string")
[1] "new␣string"
```

Special calls: replacement calls

- they act like they modify their arguments in place, and have the special name `xxx <-`
- they typically have two arguments (x and value), although they can have more, and they must return the modified object

```
`second<-` <- function(x, value) {
  x[2] <- value
  x
}
x <- 1:5
second(x) <- 0
x
[1] 1 0 3 4 5
```

- when R evaluates the assignment `second(x) <- 5`, it notices that the left hand side of <- is not a simple name, so it looks for a function named `second<-` to do the replacement
- if additional arguments are needed, they go in between x and value

```
`modify<-` <- function(x, position, value) {
  x[position] <- value
  x
}
modify(x, 1) <- -5
x
[1] -5  0  3  4  5
```


Functions : additional topics

Return values

- the last expression evaluated in a function becomes the return value

```
f <- function(x) {
  if ( x < 10 ){ 0 } else { 10 }
}
f(5)
[1] 0
```

- functions can return only a single object
- this is not a limitation because they can return a list containing any number of objects

Invisible values

- functions can return invisible values, which are not printed out by default when you call the function

```
f1 <- function() 1
f2 <- function() invisible(1)
f1()
[1] 1
f2()
```

Value not printed when calling the function but it is "returned"

```
f1() == 1
[1] TRUE
f2() == 1
[1] TRUE
```

- the most common function that returns invisibly is <-

Functions: `on.exit()` trigger

- functions can set up other triggers to occur when the function is finished using `on.exit()`
- the code inside `on.exit()` is always run, regardless of how the function exits, whether with an explicit (early) return, an error, or simply reaching the end of the function body

```
in_dir <- function(dir, code) {
   old <- setwd(dir)
   on.exit(setwd(old))
   force(code)
 }

getwd()
[1] "/Users/alberto/Documents/didattica/PhysicsOfData/R_code"

in_dir("~", getwd())
[1] "/Users/alberto"

getwd()
[1] "/Users/alberto/Documents/didattica/PhysicsOfData/R_code"
```

14.04.21

R functionals

Alberto Garfagnini

Università di Padova

Advanced R 02

Functionals basics

Definition

a FUNCTIONAL is a function that
takes FUNCTION as INPUT
and returns a VECTOR as OUTPUT

- example:

```
randomize <- function(f) f(runif(10^3))
randomize(mean)
#> [1] 0.4954407
randomize(mean)
#> [1] 0.491658

randomize(sum)
#>[1] 507.5148
```

apply function to the results of

- typical examples in base R: functionals

  `lapply()`, `apply()` and `tapply()`

- other example: `integrate()`

```
integrate(dnorm, -Inf, Inf)
#> 1 with absolute error < 9.4e-05
```

Functionals : replacement for `loops` ?

a common use of functionals is as alternative to for loops

NOTE

- `for` loops are not slow by themselves
- what makes them slow is what programmers do inside the for loop body

ex: modifying a data structure makes the loop slow because each modification creates a copy: `copy-on-modify`

functionals
for
while
repeat

- switching from loop to functional is a pattern matching exercise:
  goal: find a functional that matches the basic loop form

Our first functions: `purrr::map()`

- it takes a vector and a function
- it calls the function for each vector element
- it returns the results in a list


```
purrr::map(1:10, f)
```

vf library uploaded

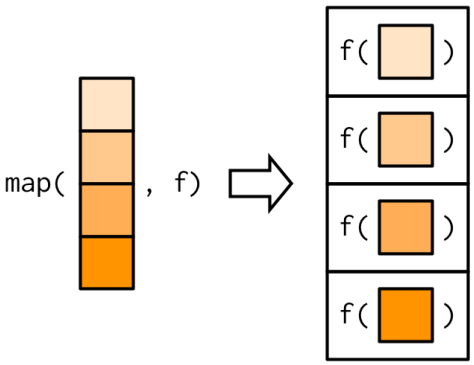
is equivalent to

```
list(f(1), f(2), ..., f(10))
```

```
double <- function(x) x*2
xd <- purrr::map(1:10, double)
str(xd)
#>List of 10
#> $ : num 2
#> $ : num 4
...
#> $ : num 18
#> $ : num 20
```

go back to the vector

```
unlist(xd)
#> [1]  2  4  6  8 10 12 14 16 18 20
```



Example 1

- we have a tibble with different data sets

```
dt <- tibble( a1 = rnorm(10), b1 = runif(10),
              c1 = rpois(10, 3.7), d1 = rbeta(10, 0.3, 5) )
```

- we want to evaluate the median of each colum

```
omed <- vector("double", ncol(dt))
omed
#> [1] 0 0 0 0
for (i in seq_along(dt)) {
    omed[[i]] <- median(dt[[i]])
}
omed
#> [1] 0.165312063 0.487255521 4.000000000 0.009203981
```

with loop

- it's possible to wrap up for loops in a function, and call that function instead of using the for loop directly

```
purrr::map_dbl(dt, median)
#>          a1          b1          c1          d1
#> 0.165312063 0.487255521 4.000000000 0.009203981
```

map ~ one line of code only

- all the `map_*()` functions use `...` to pass along additional arguments to `.f` each time it's called

returns what specified

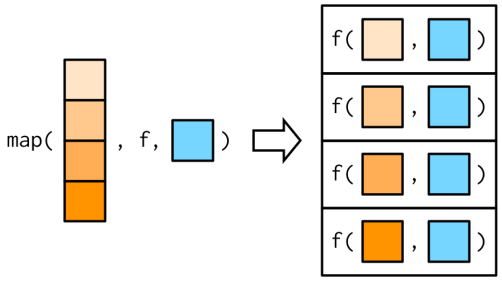
```
purrr::map_dbl(dt, mean, trim=0.5)
#>          a1          b1          c1          d1
#> 0.165312063 0.487255521 4.000000000 0.009203981
```

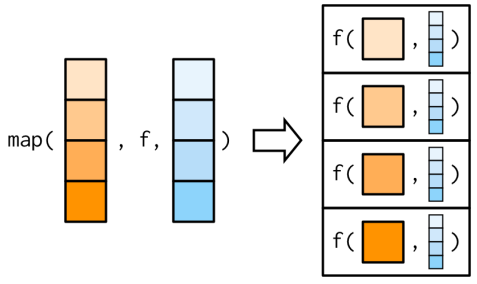

map()

the function `map()` returns a list:

```
     a list or vector
    ||
    \/
map(.x, .f, ...)
       /\   |-----> pass additional arguments to .f each
       ||           time it is called
       ||
       a function, formula or vector
```

- `map_lgl()`, `map_int()`, `map_dbl()` and `map_chr()` return a vector of specific type (logical, integer, double or character)
- `map_dfr()` and `map_dfc()` return a data frame created by row or by column
- any arguments that come after `f` in the call to `map()` are inserted after the data in individual calls to `f()`





Example 2: map()

- we generate 10 sets of random numbers from a probability distribution

```
1:10 %>% map(rnorm, n=20) -> l1
```

- this can be done using an anonyous function

```
1:10 %>% map(function(x) rnorm(n=20, x)) -> l2
```

- or by using a one-sided formula

```
1:10 %>% map( ~ rnorm(n=20, .x) ) -> l3
```

link to the first element

- there are a few shortcuts that you can use with `.f` in order to save a little typing
- `.x` and `.y` are used for two argument functions, and `..1`, `..2`, `..3`, ... for all the additional arguments

- `map()` can be chained:

```
 1:10 %>%
      map(rnorm, n=20) %>%
      map_dbl(mean)
#> [1]  0.8355395  2.0266397  3.1451209  3.9854774  5.0977312
#> [6]  6.0904780  6.9547342  8.4906865  8.9917292 10.1268192
```


Mapping over multiple arguments: map2()

- as an example we want to generate several data sets from a normal distribution with different mean and variance

```
mus <- list(5, 10, -3)
sigmas <- list(1, 5, 10)
map2(mus, sigmas, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.17 5.24 5.54 4.8 5.44
#>  $ : num [1:5] 12.71 7.01 9.56 7.25 10.74
#>  $ : num [1:5] -8.72 9.89 -14.54 3.51 -9.49
```

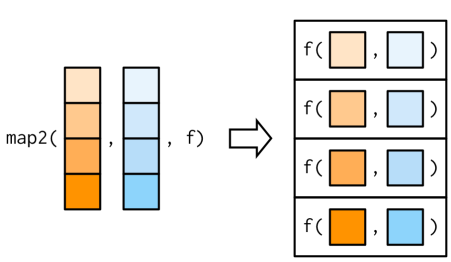
printed with this

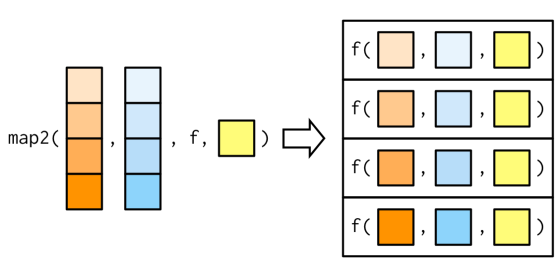
- the same results could have done iterating over indices

```
seq_along(mus) %>%
    map( ~rnorm(5, mus[[.]], sigmas[[.]]) ) %>% str()
#> List of 3
#>  $ : num [1:5] 4.73 6.52 2.68 5.42 5.35
#>  $ : num [1:5] 14.913 -0.695 11.702 17.911 1.795
#>  $ : num [1:5] -4.14 -1.08 -7.85 5.79 13.73
```

sequenza degli indici 1 ... n

- but the code with map2() is simpler and cleaner

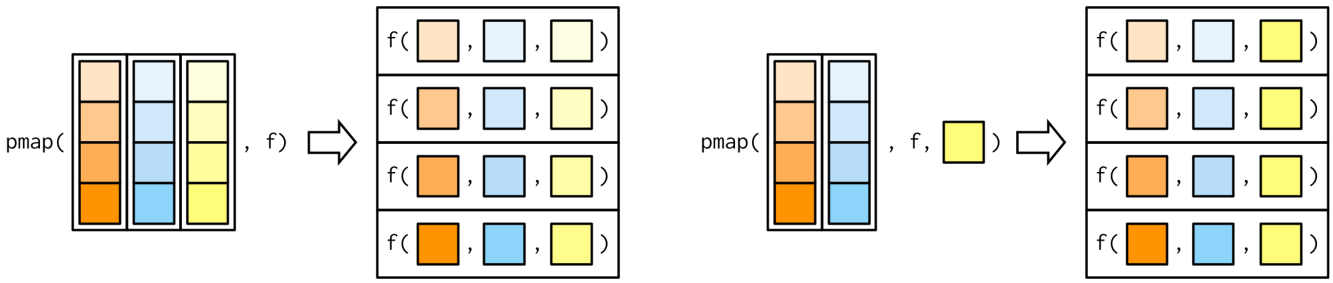




additional functions: `pmap()` and `imap()`

- in case of multiple arguments, purrr provides `pmap()` which takes a list of arguments
- if you don't name the list's elements, `pmap()` will use positional matching when calling the function. This makes the code harder to read ➜ use named arguments:

```
args2 <- list(mean = c(5, 10, -3),
              sd = c(1, 5, 10), n = 5)
args2 %>%
          pmap(rnorm) %>%
                str()
#> List of 3
#>  $ : num [1:5] 5.38 4.54 3.85 5.44 5.42
#>  $ : num [1:5] 17.038 6.072 15.107 0.697 6.488
#>  $ : num [1:5] -10.7 2.99 -1.12 17.47 13.08
```



Invoking different functions: `invoke_map()`

- a setp up in complexity is to invoke different functions with different parameters (values and meanings):

```
fgen <- c("runif", "rnorm", "rpois")

fpar <- list(
           list(min = -1, max = 1),
           list(sd = 3),
           list(lambda = 7.5))

invoke_map(fgen, fpar, n = 5) %>% str()
```

calling a list of function

```
#> List of 3
#>  $ : num [1:5] -0.7744 -0.0524 0.7523 0.5074 0.5284
#>  $ : num [1:5] 3.162 -2.766 -0.298 -2.849 -2.638
#>  $ : int [1:5] 5 7 10 6 4
```

Example: reading different files and combine in one data.frame

- our data is organized in text files accoring to different years:
- data_2020_Italy.csv, data_2021_Italy.csv
- we want to read the data and combine them in one data.frame

```
read_my_csv <- function(year, country) {
    filename <- paste0(year, "_", country, ".csv")
    mobdata_dir <- "./Region_Mobility_Report_CSVs"
    filepath <- file.path(mobdata_dir, filename)
    message(paste("Reading from file:", filepath))
    read_csv(filepath)
}

years <- 2020:2021
country <- "Italy"

mbdata <- map_df(years, read_my_csv, country)
```

```
Reading from file: ./Region_Mobility_Report_CSVs/2020_IT.csv
...
Reading from file: ./Region_Mobility_Report_CSVs/2021_IT.csv
...
```