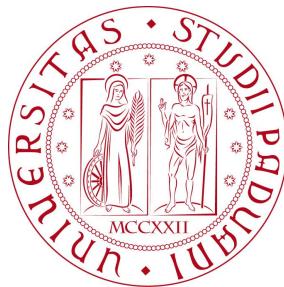


Introduction to R

Alberto Garfagnini

Università di Padova

AA 2021/2022 - Rlecture 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.
- 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
- 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 4.1.2 (Bird Hippie), released on November 1st, 2021
- A New R version, 4.1.3 (One Push-Up), will be released on March 10, 2022

R Web Resources

- R Web Site: <https://www.r-project.org/>
- R source code: <https://cran.r-project.org/src/base/R-4/>
- R Developer Page: <https://developer.r-project.org/>
- 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 22 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):
 - https://www.youtube.com/channel/UCqEdfW-1KUn_QQyQogxqLeA/
- P. Dalgaard, A Brief History of R and some Thoughts about the Future



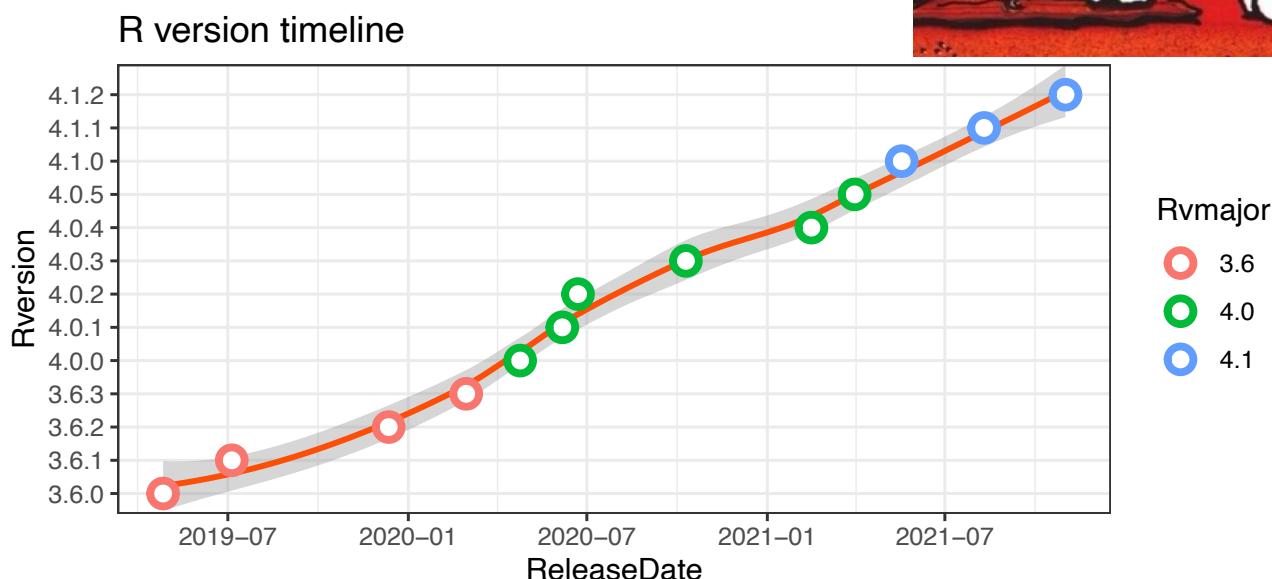
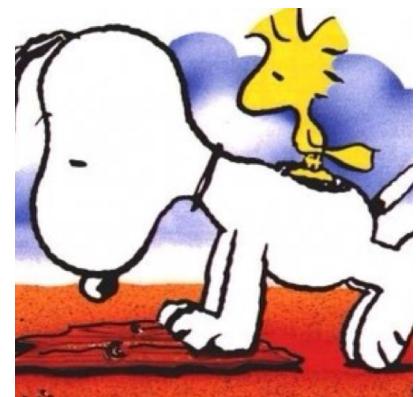
A. Garfagnini (UniPD)



2

R versions and codenames

- R release names are taken from the Peanuts comics
- R 4.1.3 (One Push-Up)
- see Peter Dalgaard talk at useR!2018
<https://www.youtube.com/watch?v=z1vTSdRolgI>



A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R01

3

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:



The R Manuals			
<i>edited by the R Development Core Team.</i>			
Manual	R-release	R-patched	R-devel
An Introduction to R is based on the former "Notes on R", gives an introduction to the language and how to use R for doing statistical analysis and graphics.	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
R Data Import/Export describes the import and export facilities available either in R itself or via packages which are available from CRAN.	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
R Installation and Administration	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
Writing R Extensions covers how to create your own packages, write R help files, and the foreign language (C, C++, Fortran,...) interfaces.	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
A draft of The R language definition documents the language <i>per se</i> . That is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions.	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
R Internals : a guide to the internal structures of R and coding standards for the core team working on R itself.	HTML PDF EPUB	HTML PDF EPUB	HTML PDF EPUB
The R Reference Index : contains all help files of the R standard and recommended packages in printable form. (9MB, approx. 3500 pages)	PDF	PDF	PDF

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R01

4

How to install R

Local Installation

- from sources: <https://cran.stat.unipd.it/src/base/R-4/R-4.1.2.tar.gz>
- or using pre-defined packages for
 - Linux (check with your favorite distribution)
 - mac OS X (from 10.13 above and for 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 4.1.2, alone https://hub.docker.com/_/r-base
 - docker pull jupyter/r-notebook:latest, for R 4.1.2 integrated with Jupyter <https://hub.docker.com/r/jupyter/r-notebook>

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R01

5

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 0xffffffff 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
```

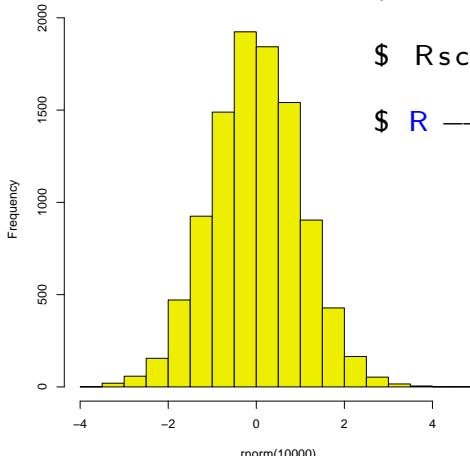
How to run R ?

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
```



Batch mode R

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

```
Histogram of rnorm(1000) $ R CMD BATCH xh_plot.R
```

```
$ Rscript xh_plot.R
```

```
$ R --no-save < xh_plot.R > xh_plot.out
```

← file: xh.pdf

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R01

8

Starting an interactive R session

- the R program can be invoked from the bash shell

```
$ R  
  
R version 4.1.2 (2021-11-01) -- "Birds of a Feather"  
Copyright (C) 2021 The R Foundation for Statistical Computing  
Platform: x86_64-apple-darwin17.0 (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.  
  
>
```

- 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

- 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

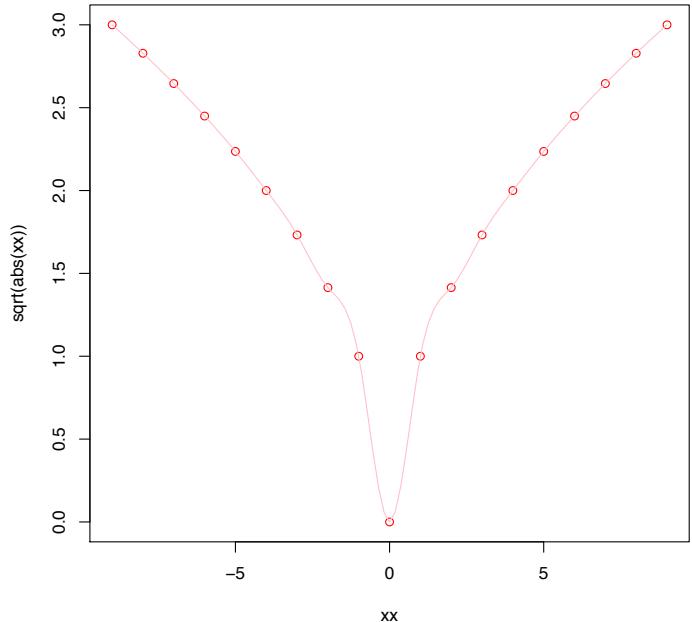
sqrt> require(graphics)

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")
```



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)

                           Information on package 'base'
Description:

Package:          base
Version:          4.0.3
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 4.0.3
Suggests:         methods
Built:             R 4.0.3; ; 2020-10-11 03:26:00 UTC; unix

Index:

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

R packages - listing

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

```
> pkg <- installed.packages()
> df_pkg <- data.frame(pkg)
> names(df_pkg)
[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] 299

> df_pkg[c(1:9,295:299),c(1,3,10,16)]
   Package Version License Built
abind      abind  1.4-5    LGPL (>= 2) 4.0.2
askpass    askpass  1.1      MIT + file  LICENSE 4.0.2
assertthat assertthat 0.2.1     GPL-3 4.0.2
backports   backports 1.2.0     GPL-2 | GPL-3 4.0.2
base64enc  base64enc 0.1-3     GPL-2 | GPL-3 4.0.2
BayesFactor BayesFactor 0.9.12-4.2      GPL-2 4.0.2
bayestestR bayestestR  0.8.0     GPL-3 4.0.2
bbmle       bbmle   1.0.23.1      GPL 4.0.2
bdsmatrix   bdsmatrix  1.3-4     LGPL-2 4.0.2
stats4      stats4   4.0.3     Part of R 4.0.3 4.0.3
survival    survival  3.2-7     LGPL (>= 2) 4.0.3
tcltk       tcltk    4.0.3     Part of R 4.0.3 4.0.3
tools       tools    4.0.3     Part of R 4.0.3 4.0.3
utils       utils    4.0.3     Part of R 4.0.3 4.0.3
```

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:

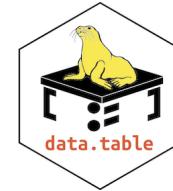
<https://cran.r-project.org/web/packages/data.table/index.html>

- on GitHub: <https://github.com/Rdatatable/data.table>

README.md

data.table

CRAN OK build passing  build passing  codecov 99% pipeline passed downloads 523K/month
Depsy 100th percentile CRAN usage 806 BioC usage 201 indirect usage 2295



`data.table` provides a high-performance version of `base R's data.frame` with syntax and feature enhancements for ease of use, convenience and programming speed.

30 January 2020

List-columns in `data.table` - Tyson Barrett, [rstudio::conf\(2020L\)](#)

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R01

16

R session housekeeping

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

```
> 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 : $\text{Arg}(x+yi) = \text{atan}(y/x)$
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
<code>log(x)</code>	base e log of x
<code>exp(x)</code>	anti-log of x
<code>log(x,n)</code>	base n log of x
<code>log10(x)</code>	base 10 log of x
<code>sqrt(x)</code>	square root of x
<code>factorial(x)</code>	$x! = x(x - 1)(x - 2) \dots 3 \cdot 2 \cdot 1$
<code>choose(n,x)</code>	binomial coefficient, $n!/(x!(n-x)!)$
<code>gamma(x)</code>	$\Gamma(x)$ for real x, $(x-1)!$ for integer x
<code>lgamma(x)</code>	natural log of $\Gamma(x)$
<code>abs(x)</code>	absolute value for x
<code>floor(x)</code>	greater integer less than x
<code>ceiling(x)</code>	smallest integer greater than x
<code>trunc(x)</code>	closest integer to x between 0 and x; it behaves as <code>floor()</code> for $x > 0$ and like <code>ceiling()</code> 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
<code>cos(x)</code>	cosine of x in radians
<code>sin(x)</code>	sine of x in radians
<code>tan(x)</code>	tangent of x in radians
<code>asin(x), acos(x), atan(x)</code>	inverse trigonometric functions for real or complex numbers
<code>asinh(x), acosh(x), atanh(x)</code>	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
```

- 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

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

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

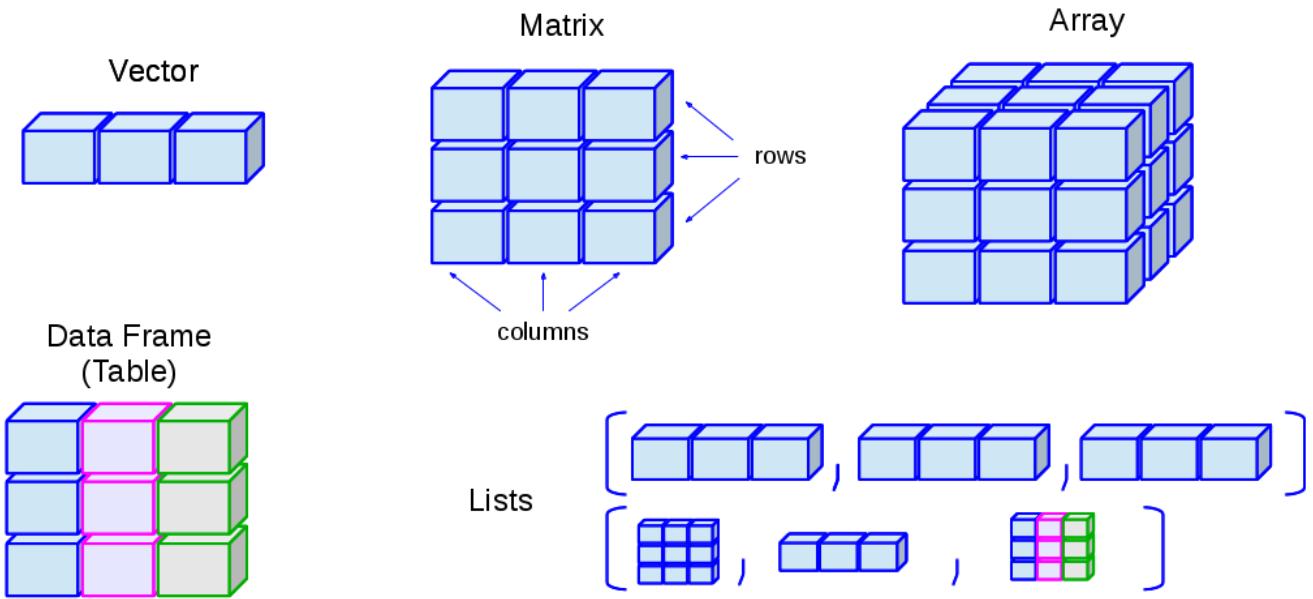
> 119 %% 12    # remainder (modulo) of the division
[1] 11

> 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

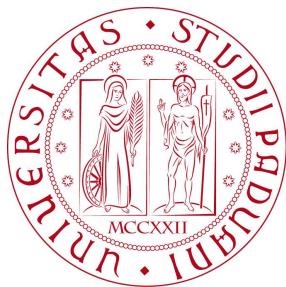


R data types: vectors

Alberto Garfagnini

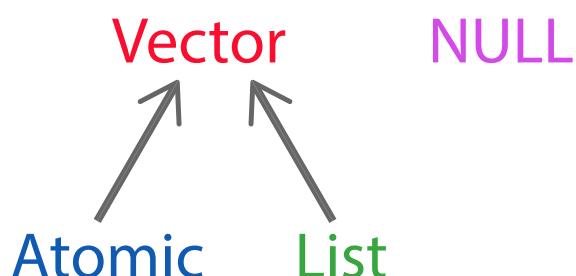
Università di Padova

AA 2021/2022 - 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

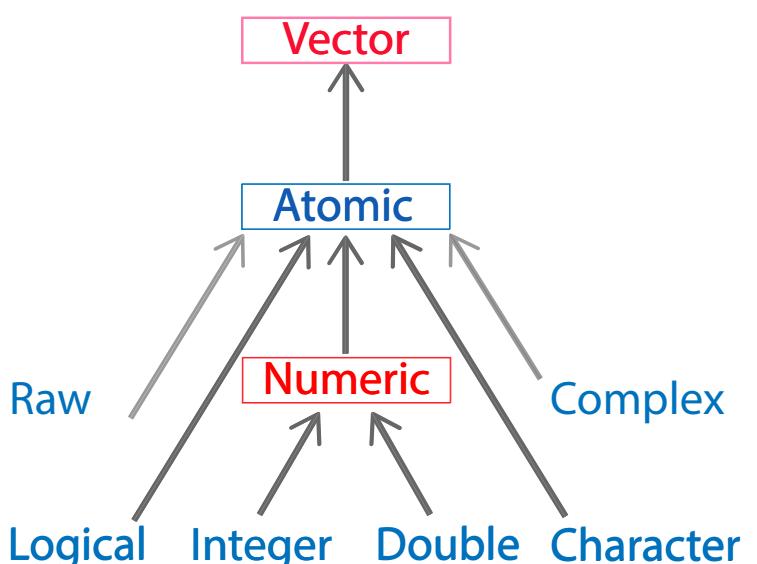


- every vector can have `attributes`
- two important attributes are: `dimension` and `class`
- `dimension` allows to create a `matrix` (`dim=2`) and `array` (`dim>2`)
- `class` powers the `S3` object system in R

Atomic Vectors

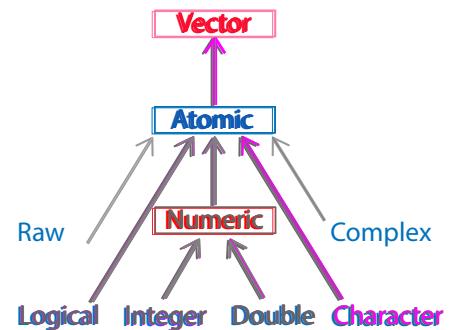
R atomic vectors

- There are 4 basic types of atomic vectors:
 - **logical** : TRUE, FALSE
abbreviated with T and F
 - **double** : 2.75 (decimal), 1.23E4
(scientific) or 0xcafe (hexadecimal)
 - **integer** : written similar to double, but with an L suffix (123L, 1E3L or 0xcafeL)
 - **character** : "a", "a word". These are strings surrounded by " or '
special characters are escaped with \
See ?Quotes for details
- 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           y <- 3L          z <- x>0           w <- 'three'
class(x)         class(y)        class(z)
%> [1] "numeric" %> [1] "integer" %> [1] "logical"
typeof(x)        typeof(y)       typeof(z)
%> [1] "double"   %> [1] "integer" %> [1] "logical"
length(x)        length(y)      length(z)
%> [1] 1          %> [1] 1        %> [1] 1
str(x)           str(y)         str(z)
%> num 3          %> int 3       %>
                                         logi true
                                         %>
                                         chr "three"
```

R vectors

- 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)
%> num [1:4] 1 2 5 8
```

- `c()` calls can be combined:

```
y <- c(y, 12, c(1, 7, 8)
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'

- 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
```

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

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

- there are functions to test whether 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
```

- 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
```

- 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

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

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
```

# Vector indexing

---

- 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
```

- 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 assigning 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 `unname()`, or setting `names(x) <- NULL`

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

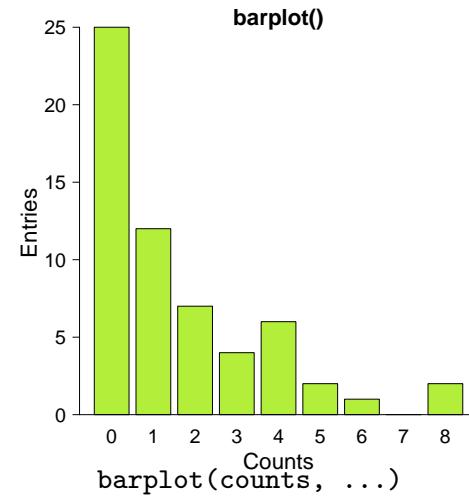
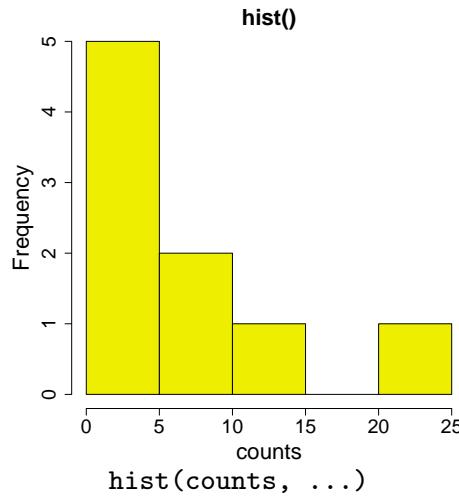
# Example : naming vector elements

- sometimes it 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 its 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 transform 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 its 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
```

- `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
```

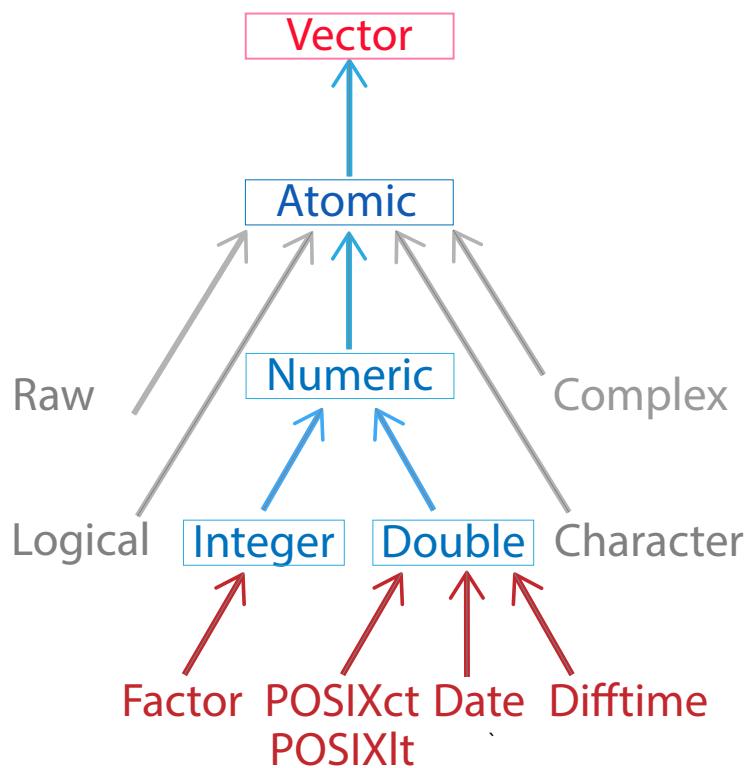
R vector functions

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

S3 Atomic Vectors

S3 atomic 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

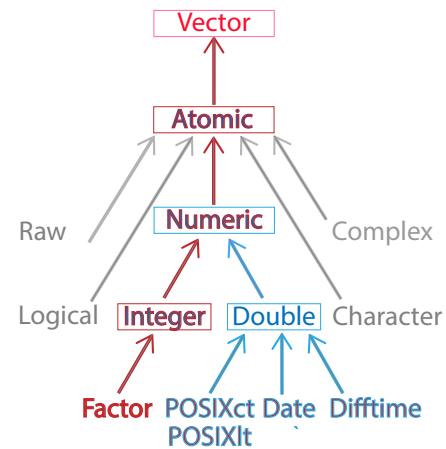


S3 atomic vectors : factors

- 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"
```



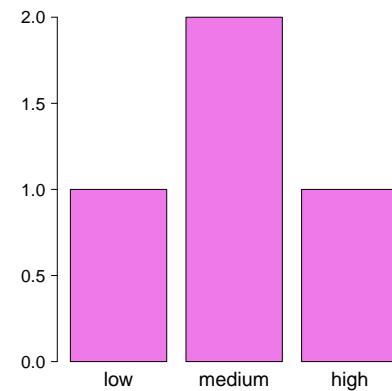
```
coord <- factor(c("Est", "West", "Est", "North"),
                  levels = c("North", "Est", "South", "West")) ; coord
%> [1] Est     West    Est     North
%> Levels: North Est South West

> 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

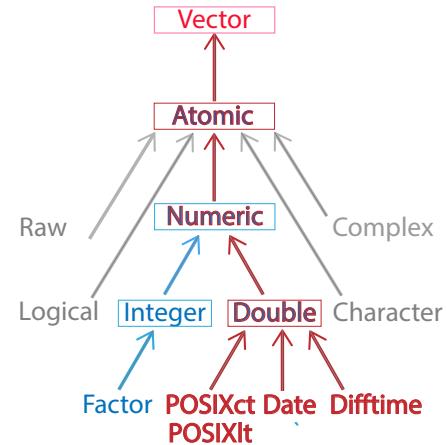
S3 atomic vectors : Dates

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(), tzon="CET")
now_ct
%> [1] "2020-03-15 14:22:41 UTC"

r20bdy_ct <- as.POSIXct("2020-02-29 12:00", tzon= "CET")
now_ct - r20bdy_ct
%> Time difference of 15.14075 days
```

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

```
structure(now_ct, tzon="Europe/Rome")
%> [1] "2020-03-15 15:30:53 CET"
structure(now_ct, tzon="Europe/Moscow")
%> [1] "2020-03-15 17:30:53 MSK"
structure(now_ct, tzon="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 only 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

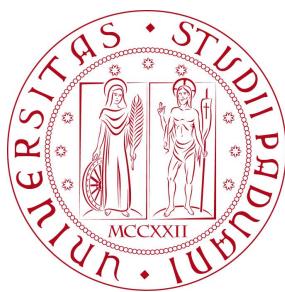
setA %in% setB
%> [1] FALSE FALSE FALSE TRUE TRUE
setA[setA %in% setB] # equal to intersect(setA, setB)
%> [1] "d" "e"
```

R flow of control and conditional execution

Alberto Garfagnini

Università di Padova

AA 2021/2022 - R lecture 3



what we have learned so far

- R is a dynamically-typed language

```
x <- 1L
y <- 2
x + y
# [1] 3

y <- 5.6
x + y
# [1] 6.6
```

- Dynamic typing allows us to assign a value of a different data type to the same variable at any time

```
y <- "Test"
x + y
# Error in x + y : non-numeric argument to binary operator
```

- But the operation is acknowledged with an error if the data types of the two variables are not compatible

Variables do not need to be declared

- variables do not need to be declared
- all types are vectors
 - there is no "scalar" type in R
 - but sometimes you have vectors of length one

Internal data representation

- floating point numbers are stored as double (64-bit)
- numbers and sequences are represented as integers

Inspecting data

Function	Description
<code>class()</code>	show the R type or storage mode of an object
<code>typeof()</code>	get/set the storage mode of an object
<code>length()</code>	get/set the length of a vector
<code>attributes()</code>	access object's attributes
<code>str()</code>	compactly display the internal structure of an R object

Floating point precision limits

- only operations among integers are exact

```
u <- sqrt(2)
u * u == 2
# [1] FALSE

u * u - 2
# [1] 4.440892e-16

print(u * u)
# [1] 2

print(u * u, digits = 18)
# [1] 2.0000000000000044
```

- the `all.equal()` function compares R objects testing 'near equality'
- the `identical()` function tests two R objects for being exactly equal

```
identical(u * u, 2)
# [1] FALSE
all.equal(u * u, 2)
# [1] TRUE
```

- check R FAQ 7.31 - Why doesn't R think these numbers are equal?

Overview

- if / else
- switch()
- for loops
- while
- ifelse()
- function()

if statement

- **if** is the typical conditional execution in many programming languages

```
x <- 35
y <- 45
if (x > y) cat("x is bigger\n")
if (y > x) cat("y is bigger\n")
# y is bigger
```

- with **else** we can build an alternative branch

```
if (x > y) {
  cat("x is bigger\n")
} else {
  cat("y is bigger\n")
}
# y is bigger
```

always us the { brackets to delimit the body

- and **if statements** can be nested

```
if (x > y) {
  cat("x is bigger\n")
} else if (y > x) {
  cat("y is bigger\n")
} else {
  cat("x and y are equal\n")
}
# y is bigger
```

switch (expression, list) conditional statement

- switch() can be used for multiple alternatives, instead of nested if / else
- expression can be a number:

```
switch(1, "red", "green", "blue", "yellow")
# [1] "red"

switch(4, "red", "green", "blue", "yellow")
# [1] "yellow"

switch(0, "red", "green", "blue", "yellow")
# NULL
switch(5, "red", "green", "blue", "yellow")
# NULL
```

expression out of range

or a string → the matching named item's value is returned

```
x <- 35
y <- 45
msg <- switch(as.character(sign(x - y)),
               "1" = "x is bigger",
               "0" = "x and y are equal",           named list
               "-1" = "y is bigger")
cat(msg, "\n")
y is bigger

as.character(sign(x - y))
# [1] "-1"
```

for loops

- Basic usage: `for (j in set_or_sequence)`

```
x <- c(1, 2, 7, 9)

for (i in x) cat(i, " | ")
# 1 | 2 | 7 | 9 |
```

- we can use a sequence 1:5

```
for (i in 1:5) cat(i, " | ")
1 | 2 | 3 | 4 | 5 |
```

- but also a decreasing sequence

```
for (i in 1:-1) cat(i, " | ")
# 1 | 0 | -1 |
```

- we can loop over a sequence generate along a vector

```
for (i in seq_along(x)) cat("( ", i, ", ", x[i], ")")
# ( 1 , 1 )( 2 , 2 )( 3 , 7 )( 4 , 9 )
```

while loops

- it's another possible construct for loops
 - while tests a condition and enters the expression only if the condition is TRUE
- syntax:

```
while (some_expression_is_true) { do_something }

i <- 1
while (i<6) {
  cat(i,"|")
  i <- i + 1
}
# 1 | 2 | 3 | 4 | 5 |
```

do - while loops in R ?

- the do - while construct does not exist in R
- it can be simulated with a repeat statement
 - while tests a condition and executes block only if the condition is TRUE : 0 to MANY
 - do - while executes at least once, and tests the condition at the end : 1 TO MANY

syntax:

```
repeat {
  statements
  if ( !condition ) { break }      condition is a logical expression
}                                     to remain in the block
```

- Here is an example

```
i <- 1
repeat {
  cat(i,"|")
  i <- i + 1
  if ( !(i<6) ) { break }
}
# 1 | 2 | 3 | 4 | 5 | >
```

Problem

- given an integer number n , compute the factorial:
- $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$

Using a `for` loop

```
n <- 5

fac <- 1
for (j in 2:n) {
  fac <- fac * j
}

cat(paste(n, " ! = ", fac, "\n"))
# 5 ! = 120
```

Problem

- given an integer number n , compute the factorial:
- $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$

Using a `while` loop

```
n <- 5

fac <- 1
tmp <- n
while (tmp > 1) {
  fac <- fac * tmp
  tmp <- tmp - 1
}

cat(paste0(n, " ! = ", fac, "\n"))
# 5 ! = 120
```

Problem

- given an integer number n , compute the factorial:
- $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$

Using a `repeat` loop

```

n <- 5

fac <- 1
tmp <- n
repeat {
  fac <- fac * tmp
  tmp <- tmp - 1
  if (tmp < 1) {
    break
  }
}

cat(paste0(n, " ! = ", fac, "\n"))
# 5! = 120

```

Example: compute the factorial

Problem

- given an integer number n , compute the factorial:
- $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 2 \cdot 1$
- we could compute the factorial using built-in functions that operate on the entire vector, avoiding the need of loops or repeats

```

cumprod(1:5)
# [1] 1 2 6 24 120

cumprod(0:5) # This does not work, '0' is in the vector
# [1] 0 0 0 0 0 0

# We could get the factorial as the maximum value of the new vector:
max(cumprod(1:5))
[1] 120

```

- R implements a built-in `factorial()` function

```

factorial(5)
# [1] 120
factorial(0)
# [1] 1
factorial(-3)
# [1] NaN
# Warning message:
# In gamma(x + 1) : NaNs produced

```

function

- functions are the most important bulding block in R

```
f <- function(x, y, z) { statement(s) }
```

- a function in R can have (a variable number of) arguments

- that may have names
- and default values

- in a call, arguments are given by position or name

```
fsum <- function(x = 1, y = 2) { x + y }

fsum()
# [1] 3

fsum(2,3)
# [1] 5
fsum(2)
# [1] 4
fsum(y=2)
# [1] 3
fsum(y=2, x=4)
# [1] 6

fsum
# function(x = 1, y = 2) { x + y }
```

R lazy evaluation

What is it?

- it's a programming strategy, used mainly in functional languages
- only necessary symbols are evaluated → only the needed objects will be loaded in memory and/or looked for

Examples

```
plaz <- function(a, b) { return 10; }

plaz()
# [1] 10
plaz(1)
# [1] 10
plaz(1, 2)
# [1] 10

play <- function(a, b) { a * 10 }

play()
# Error in play() : argument "a" is missing, with no default
play(1)
# [1] 10
play(1, 2)
# [1] 10
```

R lazy evaluation (2)

More examples

- and expression is evaluated not at the time the call is made, but only when the expression is used

```
# Here, only TRUE is evaluated
if (TRUE || no_variable) {
  12
}
# [1] 12

ping <- function(a = Sys.time(), b = Sys.time(), c = Sys.time()){
  print(a)
  Sys.sleep(1)
  print(b)
  Sys.sleep(1)
  print(c)
}

ping()
# [1] "2022-03-16 16:44:07 CET"
# [1] "2022-03-16 16:44:08 CET"
# [1] "2022-03-16 16:44:09 CET"
```

ifelse() : vectorized conditional execution

```
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
```

- we need to replace -Inf with NA

```
(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
```

`ifelse()` allows to perform `conditional execution` on the whole vector

Problem

- we generate a vector of 10^7 random numbers from a uniform distribution
- we want to **search for the maximum** value in the vector, using the R `max()` function and conventional loops

```
library(microbenchmark)

fm1 <- function(x) { max(x) }

fm2 <- function(x) {
  cmax <- x[1]
  for (i in 2:length(x)) {
    if (x[i] > cmax) cmax <- x[i]
  }
}

u <- runif(10^7)

microbenchmark(max(u), fm1(u), fm2(u))

Unit: milliseconds
expr      min       lq     mean   median      uq     max neval cld
max(u) 13.8899 14.0255 15.0040 14.3078 15.753 18.1426 100  a
fm1(u) 13.8918 13.9899 14.8993 14.4838 15.186 21.7245 100  a
fm2(u) 237.0996 238.9029 243.2218 240.7945 246.097 259.4658 100  b
```

Good/Bad practice in building vectors

Problem

- 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
}

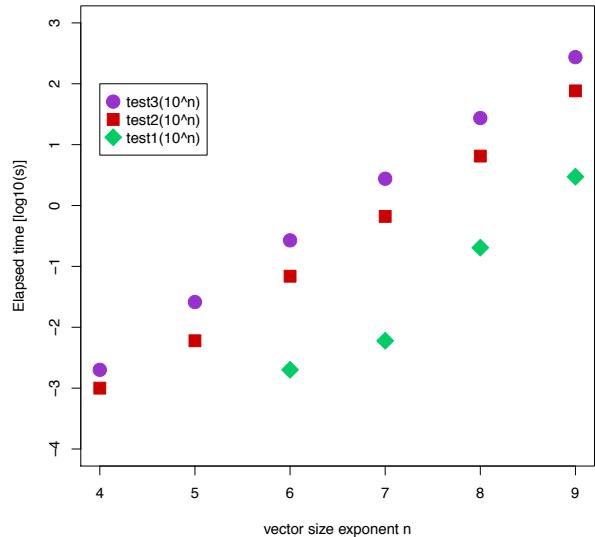
test2 <- function(n){
  y <- numeric(n)
  for (i in 1:n)
    y[i] <- i
}

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

Good/Bad practice in building vectors (2)

```
test1 <- function(n){      test2 <- function(n){      test3 <- function(n){  
    y <- 1:n                  y <- numeric(n)        y <- NULL  
}                           for (i in 1:n)          for (i in 1:n)  
                                y[i] <- i           y <- c(y,i)  
}                           }
```

- 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

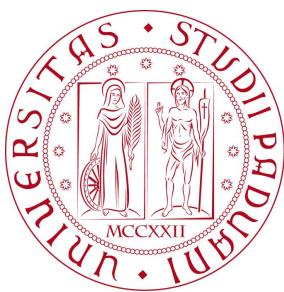


R Data Frames and Lists

Alberto Garfagnini

Università di Padova

AA 2021/2022 - R lecture 4



R internals: variables and objects creation

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

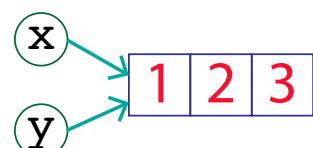
```
x <- c(1, 2, 3)
```

```
lobstr::obj_addr(x)  
"0x55d03cd66fb8"
```



- we now copy x to another variable y:

```
y <- x
```



- and modify one element of y

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

```
lobstr::obj_addr(y)  
"0x55d03dbac8c8"
```



- 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 `lobstr` package allows to visualize R data structures: it shows memory location and size of objects.

URL: <https://github.com/r-lib/lobstr>

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

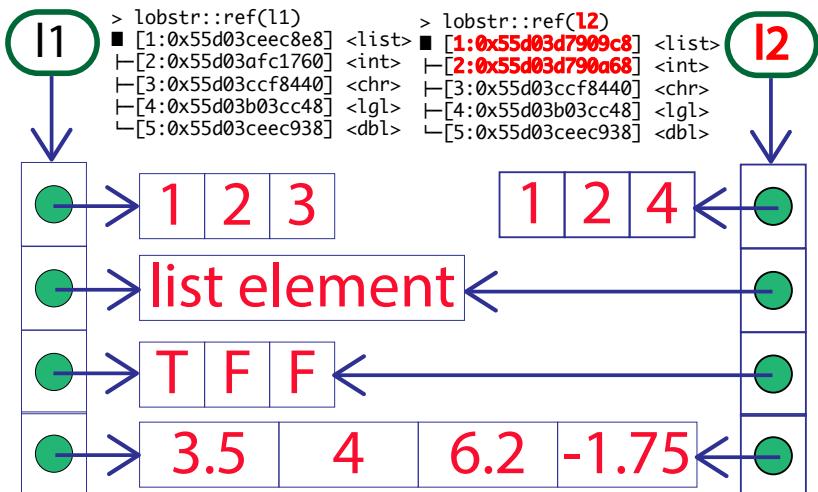
R Lists

- **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 (1)

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

- 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
```

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

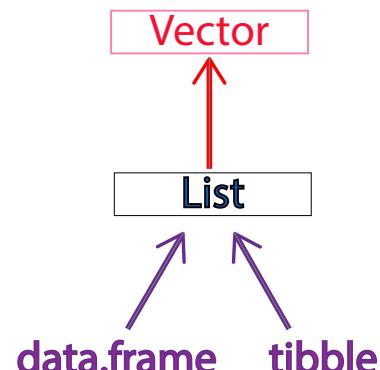
```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
# [1] "list"

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
```



R Data Frames : examples

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

Exam1	Exam2	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", "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

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" ...

```

From R 4.0

`stringsAsFactors = FALSE`
by default

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R04

8

R Data Frames objects creation (1)

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

```

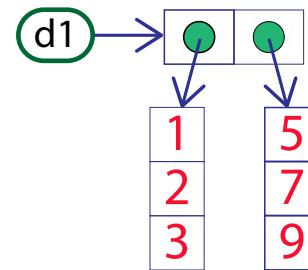
d1 <- data.frame(x = c(1, 2, 3),
                  y = c(5, 7, 9))
d1
#   x  y
# 1 1  5
# 2 2  7
# 3 3  9

```

```

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

```



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

```

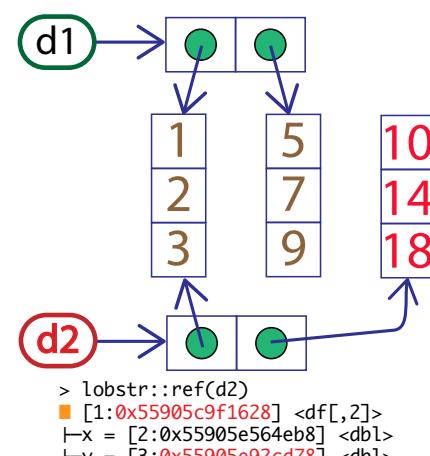
d2 <- d1
d2[, 2] <- d2[, 2] * 2
d2
#   x  y
# 1 1 10
# 2 2 14
# 3 3 18

```

```

> lobstr::ref(d2)
─ [1:0x55905c9f1628] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e92cd78] <dbl>

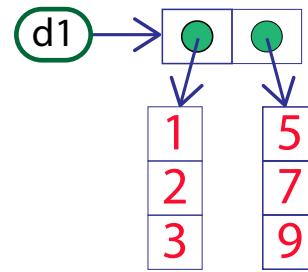
```



- 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
```

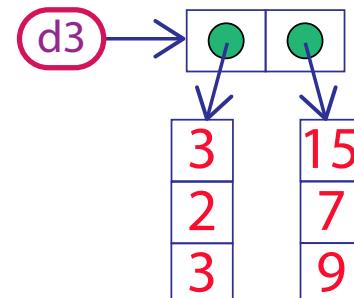
```
> lobstr::ref(d1)
─ [1:0x55905d24e9e8] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```



- 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 creates 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 ─→ 1 3 2
```

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

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

- but it is very difficult to predict 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 function is called, it makes a reference to the object → it is very difficult to predict whether 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
```

Advanced data frames : data selection (1)

- `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
```

- 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
```

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

The apply() collection functions

- are used to perform operations on all 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
- the following functions are members of the same family:
 - `lapply()`, `sapply()`, `tapply()` and `mapply()`

Function: `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
```

apply() example on data.frame

Function: `apply(X, MARGIN, FUN)`

`MARGIN=1`: manipulate rows

`MARGIN=2`: manipulate columns

```
df <- data.frame(exam1 = c(27,28,24,24,30,26),
                  exam2 = c(25,30,26,24,30,30),
                  labrep = c(29,30,29,27,29,30))
str(df)
# 'data.frame': 6 obs. of 3 variables:
# $ exam1 : num 27 28 24 24 30 26
# $ exam2 : num 25 30 26 24 30 30
# $ labrep: num 29 30 29 27 29 30
```

we compute the average mark over `exam2` and `labrep`:

```
row_avg <- apply(X = df[, 2:3], MARGIN = 1, FUN = mean)

# [1] 27.0 30.0 27.5 25.5 29.5 30.0
```

and the mean mark over `exam2` and `labrep`:

```
exam_avg <- apply(X = df[, 2:3], MARGIN = 2, FUN = mean)

# exam2 labrep
# 27.5   29.0
```

lapply() example on list

Function: `lapply(X, FUN, ...)`

```
plants <- list(height = runif(10, min = 10, max = 20),
                 mass = runif(10, min = 5, max = 10),
                 flowers = sample(1:10, 10))
str(plants)
# List of 3
# $ height : num [1:10] 16.8 11.7 13.8 20 15.1 ...
# $ mass   : num [1:10] 7.53 8.53 9.36 9.76 9.67 ...
# $ flowers: int [1:10] 9 2 8 5 4 10 6 3 1 7
```

we need to compute the average for each list element

```
lapply(plants, mean)
# $height
# [1] 15.63694
# $mass
# [1] 8.008216
# $flowers
# [1] 5.5
```

the `output of lapply()` is also a list

with `unlist()` we can "cast-it-down" to a simple named vector

```
unlist(lapply(plants, mean))
#      height      mass     flowers
# 15.636939 8.008216 5.500000
```

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
```

Overview

- `lapply()` returns a list
 - it may require second step collapsing list
- `sapply()` is a simplified version
 - it returns, in general, a vector
 - it is more specific, and less general
- more specialized functions exist:
 - `vapply()` can specify a return object
 - `mapply()` can use multiple arguments
 - `rapply()` can be used recursively

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
```

```
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

```
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 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

- 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
```

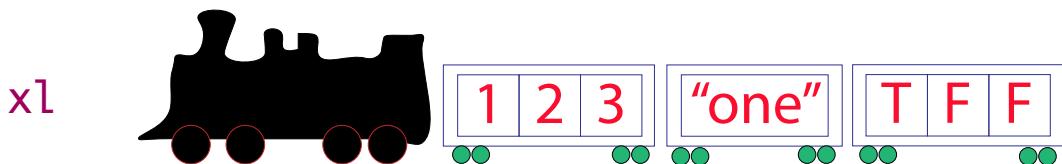
- there are two other subsetting operators:
 - `[]` is used to extract single items
 - `$` is used as a shorthand: `x$y` stands for `x[["y"]]`
- `[]` is most important while working with lists: subsetting a list with single `[]` always returns a smaller list

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

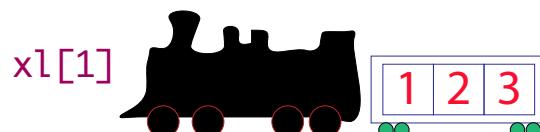
```
xl <- list(1:3, "one", c(T,F,F))
```



Selecting a single element



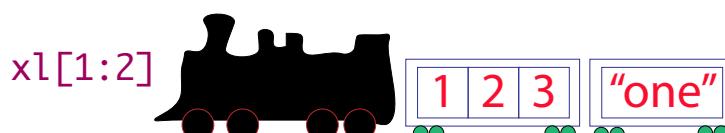
- two options are available when extracting a single element:
 - create a smaller train, with fewer cars (using `[]`)



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



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

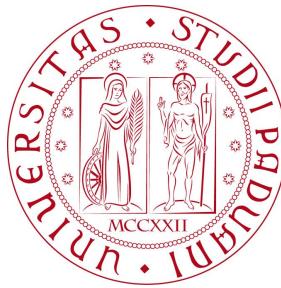


R graphics

Alberto Garfagnini

Università di Padova

AA 2021/2022 - 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 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(1000))
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`

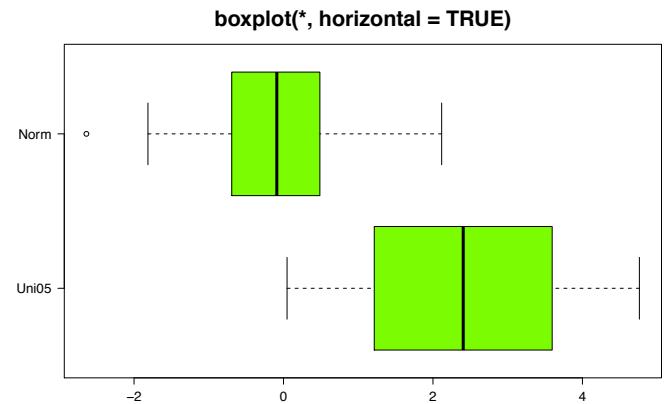
- a data visualization package created by [Hadley Wickham](#) in 2005
- it implements L. Wilkinson's Grammar of Graphics: a general scheme for data visualization which breaks up graphs into semantic components such as scales and layers

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

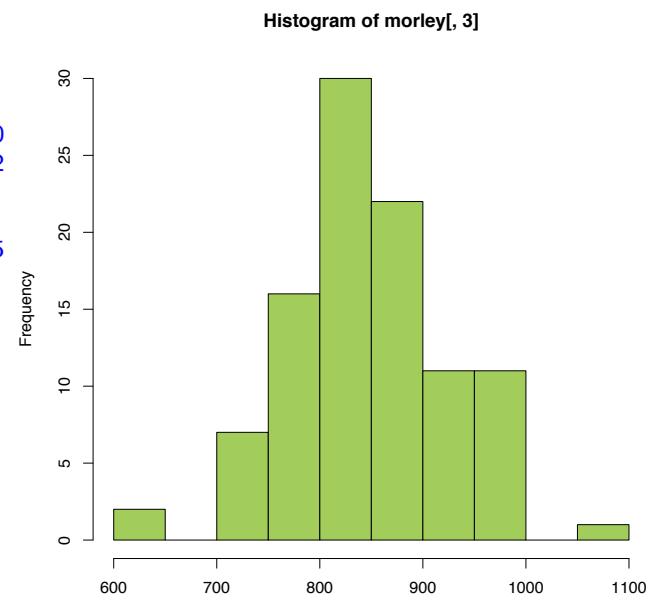
# $density
# [1] 0.0004 0.0000 0.0014 0.0032 0.0060
# [6] 0.0044 0.0022 0.0022 0.0000 0.0002

# $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

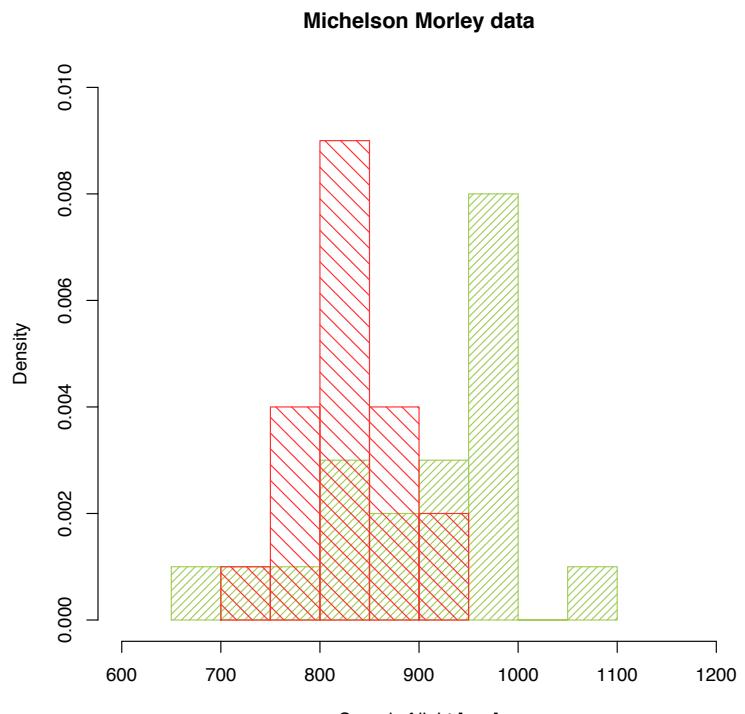
- the `add = TRUE` histogram parameters will do the job
- the option `freq = TRUE` will ensure that, in case of non equal number of observations, the heights of an interval remain the same

```
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)
```

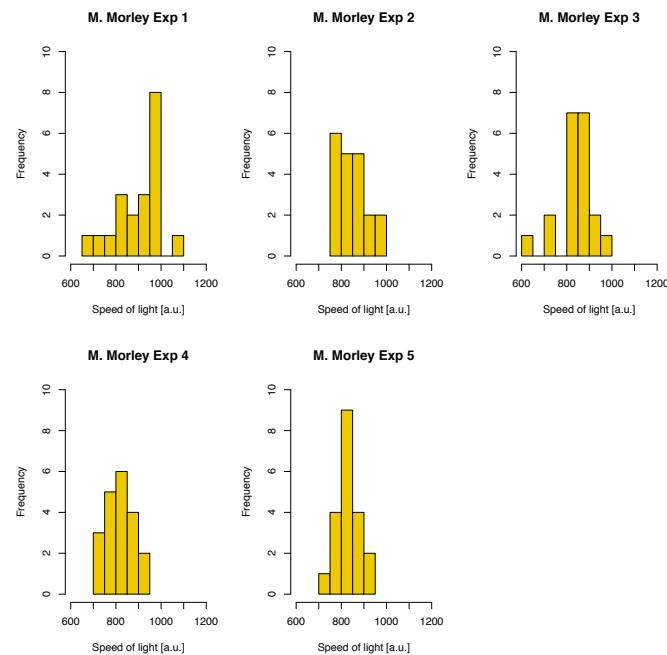


More plots on one page

- 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)  
}
```



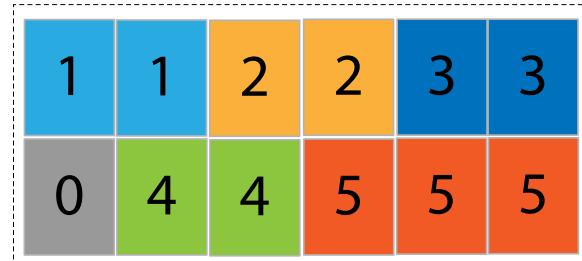
A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R05

8

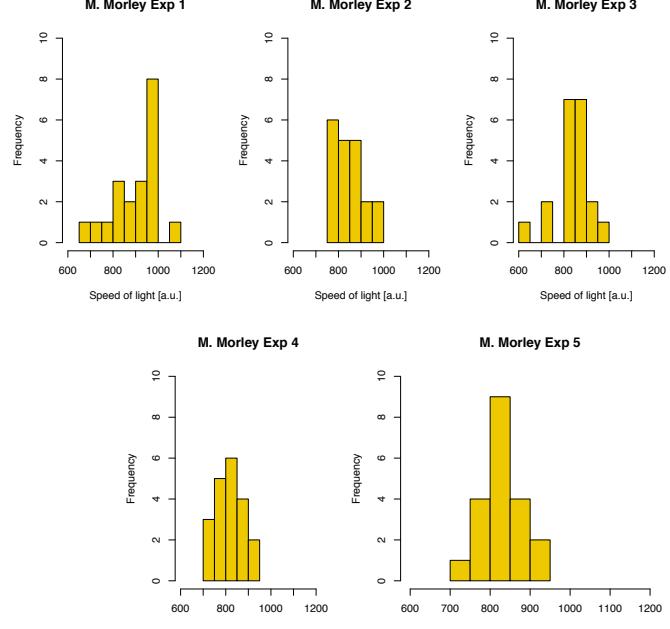
More plots on the page with layout()

- 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)  
}
```



A. Garfagnini (UniPD)

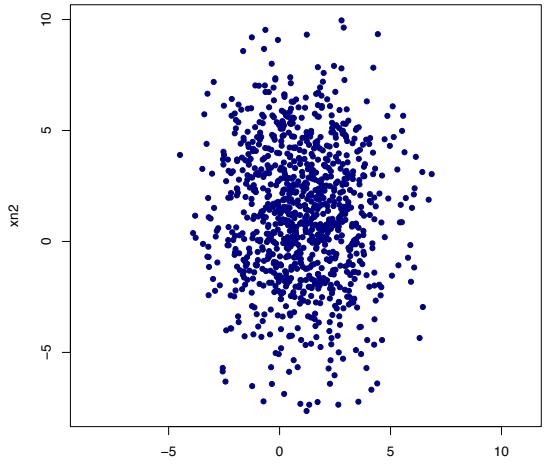
AdvStat 4 PhysAna - R05

9

The R scatter plot() function

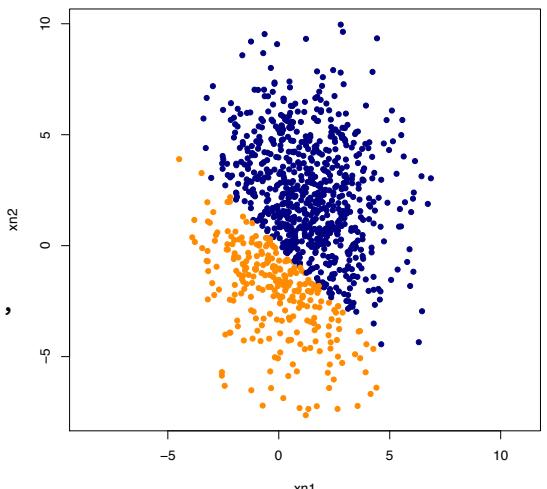
- the `plot()` function allows to produce a scatter plot of one variable versus the other
- the `asp = value` parameter allows to keep the y/x aspect ratio to a fixed value
 - `asp=1` sets the same scale for both x and y axis, even if the plot window is re-scaled

```
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)
```

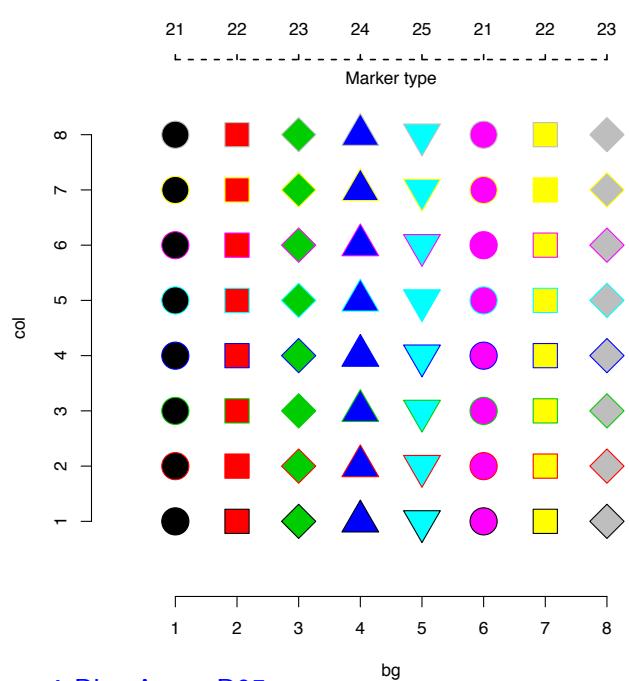


Colors for symbols and axes

- 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")
```

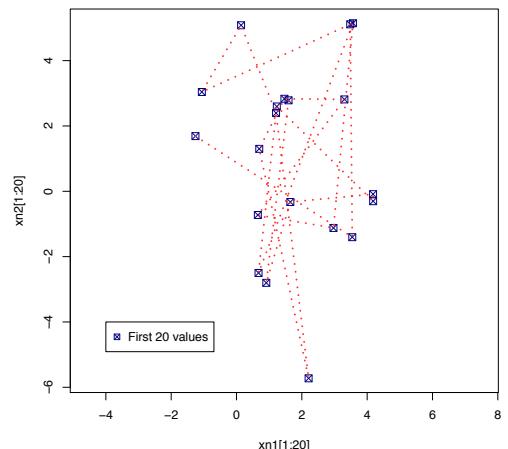


Joining points with `lines()`

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

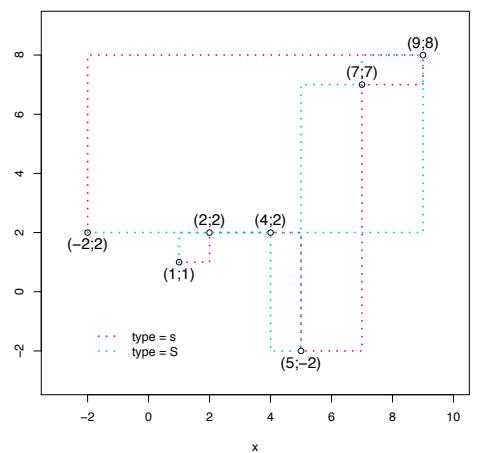
```
plot( xn1[1:20], xn2[1:20],
      pch = 7, cex=1.25, col = "navy")

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)
```



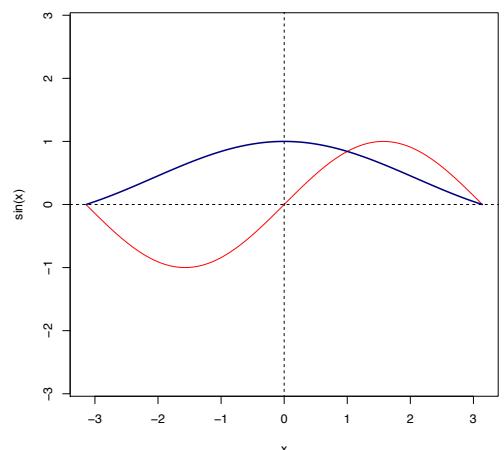
Plotting curves

- `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)

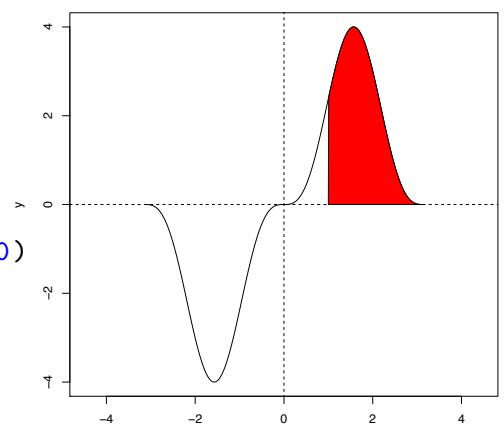
# The x-axis, going through (0,0)
abline(0,0,lty=2)
# The y-axis, going through (0,0)
abline(0,10000,lty=2)

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)
```

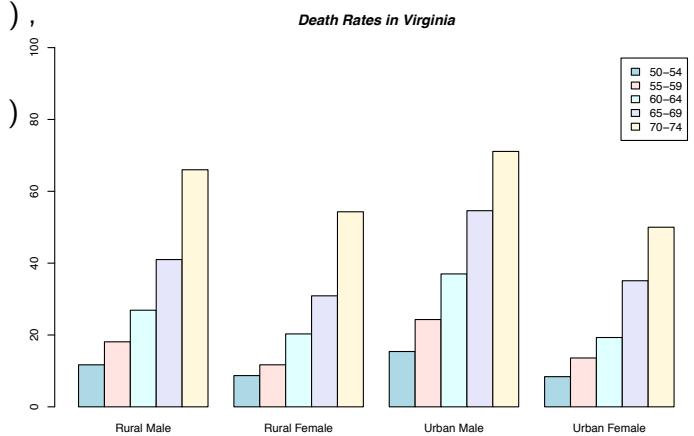


barplot

- 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

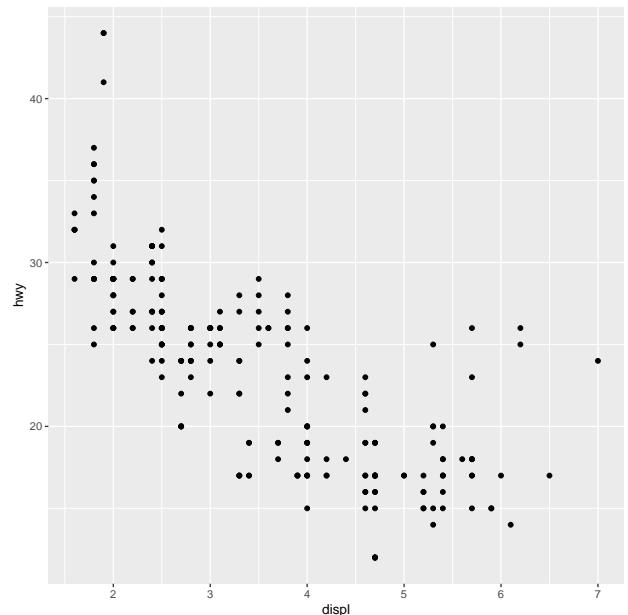
- every plot has **three components** :

- 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**

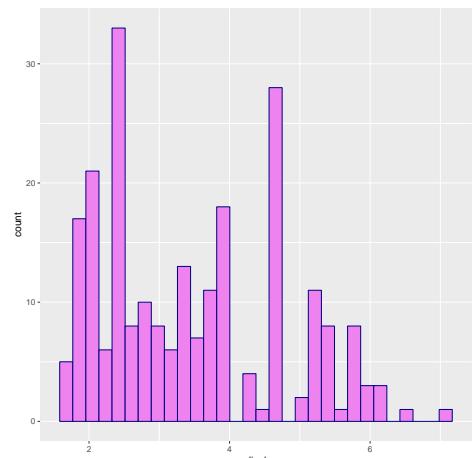
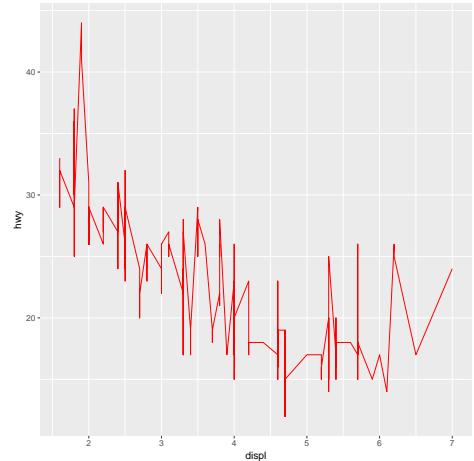


Important : data and aesthetic mappings are supplied in `ggplot()`, additional layers are added on with the '+' operator

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

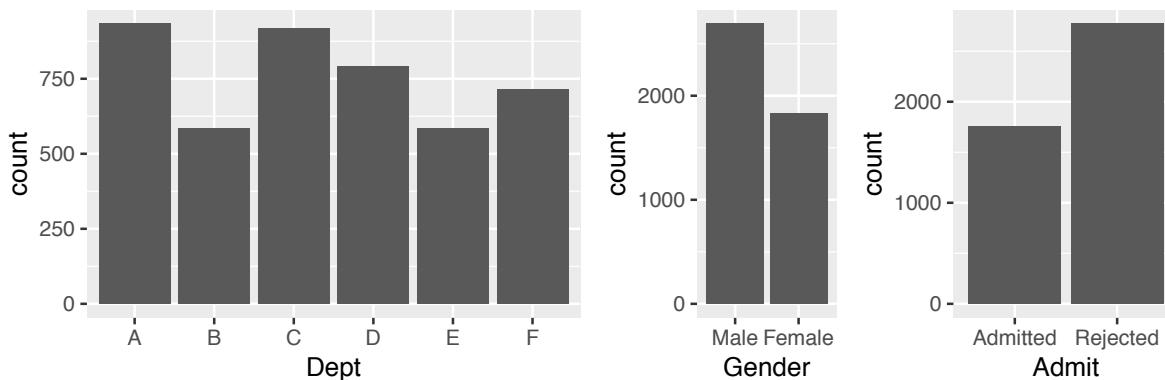
- the `UCBAdmissions` data set contains data on applicants to graduate school at Berkeley for the six largest departments in 1973 classified by admission and sex
- we create separate barcharts for the variables: department, gender, and admitted or rejected

```
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 3rd of April or the 4th 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 1st, 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 starting 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:

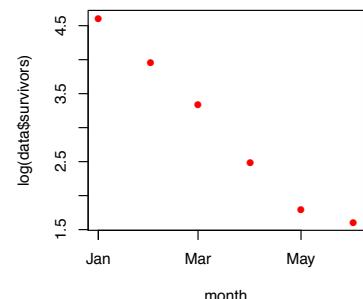
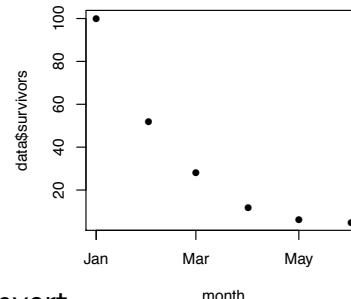
```
seq(as.POSIXlt("2019-04-01"), as.POSIXlt("2019-04-12"), length=3)
# [1] "2019-04-01 00:00:00 CEST" "2019-04-06 12:00:00 CEST"
# [3] "2019-04-12 00:00:00 CEST"
```

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)
```

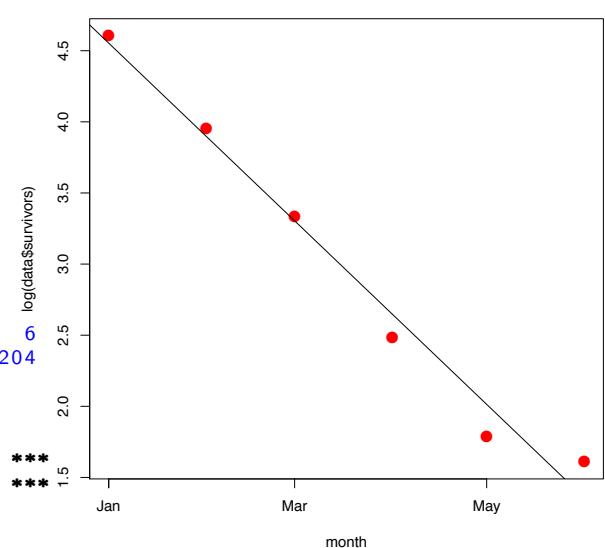
- Call:
`lm(formula = log(data$survivors) ~ dc)`

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 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
```

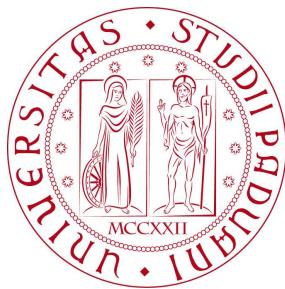
- it expands the mathematical operations to be performed with date-time objects:
3 new time span classes (`durations`, `periods` and `intervals`, borrowed from <http://joda.org>)

R data I/O

Alberto Garfagnini

Università di Padova

R lecture 6



Data Input

- numbers can be `inputed` 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\u2014Harwell",...: 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>

The readr package

- is part of the core [Tidyverse](#)
- `readr` supports seven file formats with seven `read_` functions:
 - `read_csv()`: comma separated (CSV) files
 - `read_tsv()`: tab separated files
 - `read_delim()`: general delimited files
 - `read_fwf()`: fixed width files
 - `read_table()`: tabular files where columns are separated by white-space.
 - `read_log()`: web log files

```
readr::read_delim("yield.txt", delim='\t')
%- Column specification -----
cols(
  year = col_double(),
  wheat = col_double(),
  barley = col_double(),
  oats = col_double(),
  rye = col_double(),
  corn = col_double()
)
```



<https://readr.tidyverse.org/>

Data Input from the Web and from DB

- R can read data from 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

The ECDC website homepage. The main content area discusses the COVID-19 pandemic and includes a link to 'Latest situation update, epidemiological curve and global distribution'.

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R06

8

Example: data Input from the Web

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

```
url <- "https://opendata.ecdc.europa.eu/"
fname <- "covid19/nationalcasedeath_eueea_daily_ei.xlsx"
target <- paste(url, fname, sep="")
message("target:", target)

tmp_file <- tempfile("data", "/tmp", fileext=ext)

tmp <- curl::curl_download(target, destfile=tmp_file)

data <- readxl::read_xlsx(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
```

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - R06

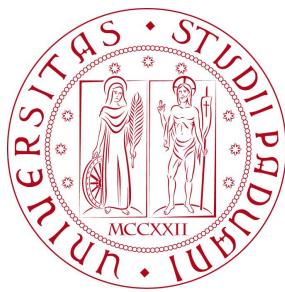
9

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 function finds the values associated with the name

```
mysum <- function(x, y) {  
  # Compute the sum of 2 vectors  
  x + y  
}  
  
> formals(mysum)           body(mysum)  
#> $x                      #> {  
#> $y                      #>     x + y  
#> }  
  
environment(mysum)  
#> <environment: R_GlobalEnv>
```

- functions, as objects, can have attributes

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

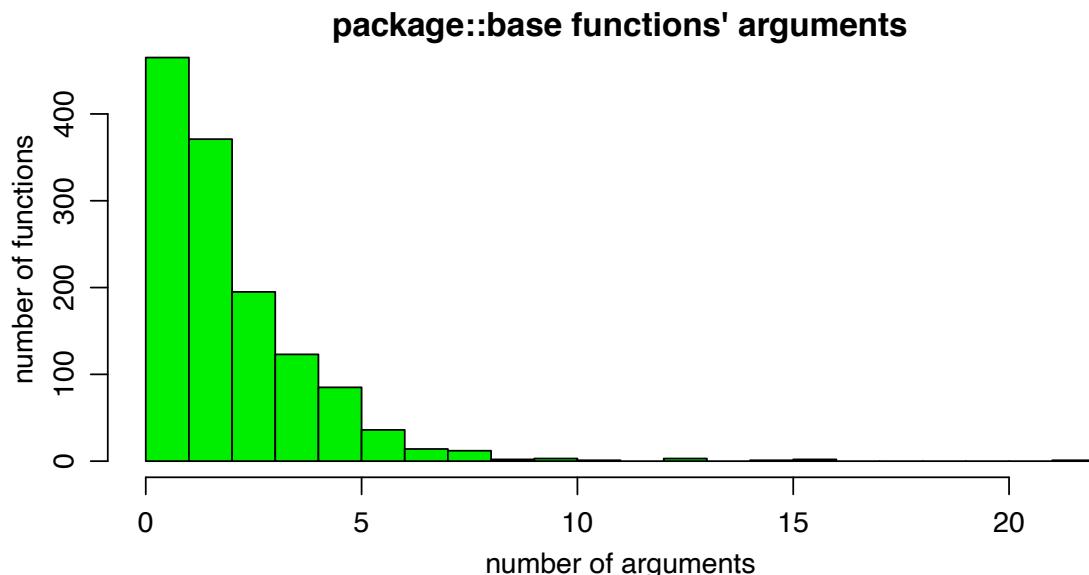
Exercise

- 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)
```

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

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



A. Garfagnini (UniPD)

AdvStat 4 PhysAna - RAdv01

4

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
```

A. Garfagnini (UniPD)

AdvStat 4 PhysAna - RAdv01

5

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
```

- $x \mid > f()$ is equivalent to $f(x)$
- $x \mid > f(y)$ is equivalent to $f(x, y)$

Lazy evaluation

- all function arguments are **lazy evaluated**

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

hstop(1)
#> [1] 10

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

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, like $x + y$ 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
```

- 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
```

- 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()`

```
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()  
[1] TRUE  
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"
```

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
```

- typical examples in base R:

`lapply()`, `apply()` and `tapply()`

- other example: `integrate()`

```
integrate(dnorm, -Inf, Inf)  
#> 1 with absolute error < 9.4e-05
```

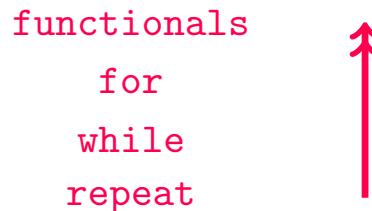
Functionals : replacement for loops ?

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**



- 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

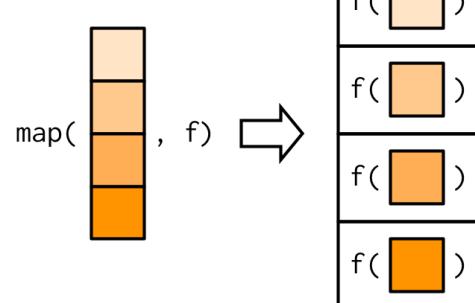
`purrr::map(1:10, f)`

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
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 column

```
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
```

- 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
```

- all the `map_*`() functions use ... to pass along additional arguments to .f each time it's called

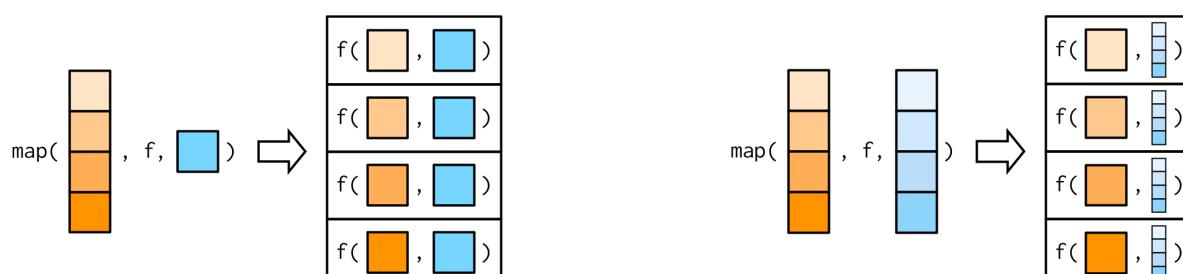
```
purrr::map_dbl(dt, mean, trim=0.5)
#>      a1          b1          c1          d1
#> 0.165312063 0.487255521 4.000000000 0.009203981
```

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 anonymous 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
```

- 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()`

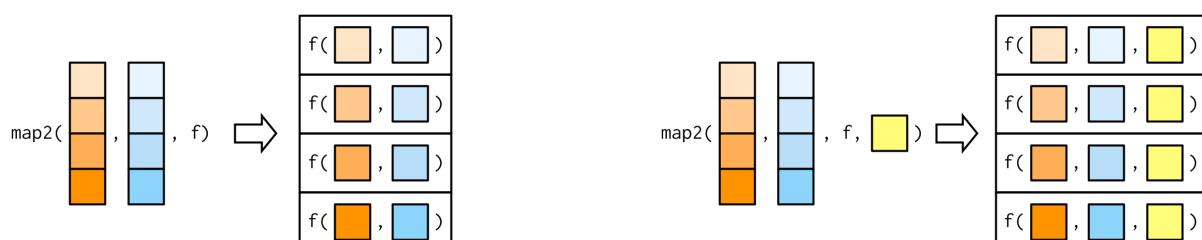
- as an example we want to generate several data sets from a normal distribution with different mean and variance

```
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
```

- 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
```

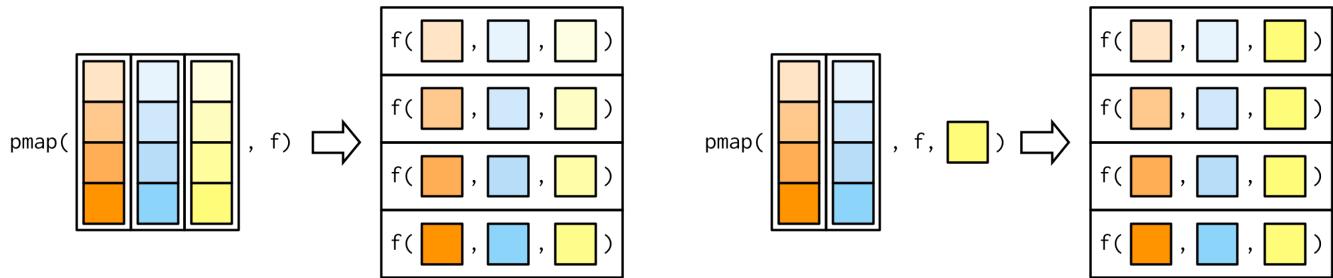
- but the code with `map2()` is simpler and cleaner



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 step 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()

#> 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
```

- our data is organized in text files according 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_CSJs"  
  filepath <- file.path(mobdata_dir, filename)  
  message(paste("Reading from file:", filepath))  
  read_csv(filepath)  
}  
  
years <- 2020:2021  
country <- "Italy"  
  
mobdata <- map_df(years, read_my_csv, country)  
  
Reading from file: ./Region_Mobility_Report_CSJs/2020_IT.csv  
...  
Reading from file: ./Region_Mobility_Report_CSJs/2021_IT.csv  
...
```