R – a short introduction

Dr. Marco Vicentini
info@marcovicentini.it

Padova, aprile 2009
Facoltà di Psicologia, Università di Padova
R programming language is a lot like magic, except instead of spells you have functions.
SPSS and SAS users are like muggles. They are limited in their ability to change their environment. They have to rely on algorithms that have been developed for them. The way they approach a problem is constrained by how SAS/SPSS employed programmers thought to approach them. And they have to pay money to use these constraining algorithms.
R users are like wizards. They can rely on functions (spells) that have been developed for them by statistical researchers, but they can also create their own. They don’t have to pay for the use of them, and once experienced enough, they are almost unlimited in their ability to change their environment.
Summary

- R: what is it? and what isn’t it?
- Introduction and basic concepts
- Basics of R
  - Syntax, Data types, etc.
- Loops and conditional structures
- Graphics
- Generating data
- Writing your own functions
References (short)

- `> help()`
- [http://cran.r-project.org/doc/manuals/R-intro.pdf](http://cran.r-project.org/doc/manuals/R-intro.pdf)
Web References

- [http://www.r-project.org](http://www.r-project.org)
The official R home page
- [http://wiki.r-project.org](http://wiki.r-project.org)
A great place loaded with information about R
R newsletter for users and developers of R code
Lots of information geared to someone new to R
- [http://addictedtor.free.fr/graphiques](http://addictedtor.free.fr/graphiques)
R graph gallery that shows off R’s graphing capability
R (programming language)
http://en.wikipedia.org/wiki/R_%28programming_language%29

S (programming language)
http://en.wikipedia.org/wiki/S_programming_language

S-PLUS
- R Reference Card (Short)
  [http://cran.r-project.org/doc/contrib/Short-refcard.pdf](http://cran.r-project.org/doc/contrib/Short-refcard.pdf)
- R Reference Card (Baron)
  [http://cran.r-project.org/doc/contrib/refcard.pdf](http://cran.r-project.org/doc/contrib/refcard.pdf)
- The R language – a short companion
  [http://cran.r-project.org/doc/contrib/R_language.pdf](http://cran.r-project.org/doc/contrib/R_language.pdf)
- R Functions For Regression Analysis
What R is and what it is not

- R is
  - a programming language
  - a statistical package
  - an interpreter
  - Open Source

- R is not
  - a database
  - a collection of “black boxes”
  - a spreadsheet software package
  - commercially supported
What R is

- Powerful tool for data analysis and statistics
  - Data handling and storage: numeric, textual
  - Powerful vector algebra, matrix algebra
  - High-level data analytic and statistical functions
  - Graphics, plotting

- Programming language
  - Language “built to deal with Real numbers”
  - Loops, subroutines
  - Classes (“OO”)
What R is not

- is not a database, but connects to DBMSs
- has no click-point user interfaces, but connects to Java, TclTk
- language interpreter can be very slow, but allows to call own C/C++ code
- no spreadsheet view of data, but connects to Excel/MsOffice
- no professional / commercial support
Pros and cons

Advantages

• Fast and free.
• State of the art: Statistical researchers provide their methods as R packages. SPSS and SAS are years behind R!
• 2nd only to MATLAB for graphics.
• Large amount of functions available in contributed packages; users can write more in FORTRAN or C/C++
• Active user community
• Excellent for simulation, programming, computer intensive analyses, etc.
• Forces you to think about your analysis.
• Interfaces with database storage software (SQL)

Disadvantages

• Not user friendly @ start - steep learning curve, minimal GUI (especially under Win/Linux, not on Mac).
• interpreted, not compiled
• No commercial support; figuring out correct methods or how to use a function on your own can be frustrating.
• Easy to make mistakes and not know.
• Working with large datasets is limited by RAM
• Data prep & cleaning can be messier & more mistake prone in R vs. Excel, and SPSS
• Some users complain about hostility on the R listserv (also I ... :D )
R and statistics

- Packaging: a crucial infrastructure to efficiently produce, load and keep consistent software libraries from (many) different sources / authors

- Statistics: most packages deal with statistics and data analysis

- State of the art: many statistical researchers provide their methods as R packages
Installation
To obtain and install R on your computer
- Go to [http://cran.r-project.org/mirrors.html](http://cran.r-project.org/mirrors.html) to choose a mirror near you
- Click on your favorite operating system (Linux, Mac, or Windows)
- Download and install the “base”

To replace the basic code editor provided by Rgui
- [http://www.sciviews.org/Tinn-R](http://www.sciviews.org/Tinn-R)
- [http://notepad-plus.sourceforge.net](http://notepad-plus.sourceforge.net)

To install additional packages
- Start R on your computer
- Choose the appropriate item from the “Packages” menu
What We Will Learn

- How to Download R from CRAN
- How to Install the Windows Version of R
- How to Extend R Functionality Using R Packages
Download R - Overview

- Browse to [http://www.r-project.org](http://www.r-project.org)
- Select the CRAN link
- Select the link for University of Padova, Italy [http://cran.stat.unipd.it/](http://cran.stat.unipd.it/)
- Locate and click the Windows link
- Select the base link
- Press the [R-2.x.x-win32.exe](http://www.r-project.org/R-2.x.x-win32.exe) link to begin the download
Downloading R

(2 of 6)
Downloading and Installing

- Downloading R (3 of 6)
Downloading and Installing

- Downloading R

(4 of 6)
Downloading and Installing

- **Downloading R**

![Image of R Archive Network page](image)

<table>
<thead>
<tr>
<th>CRAN</th>
<th>Mirrors</th>
<th>What's new?</th>
<th>Task Views</th>
<th>Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>About R</td>
<td>R Homepage</td>
<td>software</td>
<td>R Sources</td>
<td>R Binaries</td>
</tr>
<tr>
<td>Packages</td>
<td>Other</td>
<td>Documentation</td>
<td>Manuals</td>
<td>FAQs</td>
</tr>
</tbody>
</table>

---

**R for Windows**

This directory contains binaries for a base distribution and packages to run on 386/64 Windows.

Note: CRAN does not have Windows systems and cannot check these binaries for viruses. Use the normal precautions with downloaded executables.

Subdirectories:

- `base`
  - Binaries for base distribution (managed by Duncan Murdoch)
- `contrib`
  - Binaries of contributed packages (managed by Uwe Ligges)

Please do not submit binaries to CRAN. Package developers might want to contact Duncan Murdoch or Uwe Ligges directly in case of questions / suggestions related to Windows binaries.

You may also want to read the R FAQ and R for Windows FAQ.

Last modified: April 4, 2004, by Friedrich Leisch
Downloading and Installing

- Downloading R (6 of 6)
Installing R - Overview

- Locate and run the download file
- Bypass the security dialog, press Run
- Select your Language
- Continue at the Setup Wizard Welcome dialog
- Review the GNU General Public License
- Enter a directory in which to install R
Installing R

- Ensure all components are selected
- Select default Startup Options
- Use the default R folder
- Select desired Tasks (leave to Default)
- Click Finish to complete the install
Downloading and Installing

- Installing R (2 of 4)
Downloading and Installing

- Installing R

(4 of 4)
A Short History of R

- Statistical programming language “S” developed at Bell Labs since 1976 (at the same time as UNIX)
  - Statistical Computing Language
  - Originates at Bell Labs in Mid-1970’s
- Intended to interactively support research and data analysis projects
- S-PLUS
  - Exclusively licensed to Insightful (“S-Plus”)
  - Commercial Package
  - Developed in 1988 to Current
A Short History of R

- “R”: Open source platform similar to S
  - R is a Dialect of S
  - Developed by R. Gentleman and R. Ihaka (University of Auckland, NZ) during the 1990s
  - GNU-GPL licence (open source software)
  - Most S-plus programs will run on R without modification!
- Actually supported by the “R Foundation for Statistical Computing” (Vienna, Austria)
Getting started

- Call R from the shell:
  ```
  user@host$ R
  ```
- Leave R, go back to shell:
  ```
  >q()
  ```
Save information (y/n/q)? n
First Steps
Review RGui

License
- Understand your license to use and copy R
- Run and review: license(), RShowDoc("COPYING")

Citation:
- Please cite R in your publications
- Run and Review: citation()

Walk through of the R GUI menus
- Getting Help
- Working with Packages
- Other features
> 5 + (6 + 7) * pi^2
[1] 133.3049
> log(exp(1))
[1] 1
> log(1000, 10)
  [1] 3
> Sin(pi/3)^2 + cos(pi/3)^2
Error: couldn't find function "Sin"
> sin(pi/3)^2 + cos(pi/3)^2
 [1] 1
Our First R Session

- Quick Review of Vectors
  - Simple Plot
    - `x <- 1:5`
    - `y <- c(1, 2, 3, 4, 5)`
    - `plot(x, y); lines(x, y)`
  - Vectors are Objects
  - Numeric vectors are the simplest data structure available in R
  - R operates using objects and functions
Our First R Session

- **Code Snippet Deconstruction**
  ```r
  x <- 1:5
  y <- c(1, 2, 3, 4, 5)
  plot(x, y); lines(x, y)
  ```

- **Three Code Elements**
  - Objects \((x, y)\)
  - Operators \(\:"\:, \"<-\:, \";\:)\)
  - Functions \(c(), \text{plot()}, \text{lines()}\)
Let’s Take it One Step Further

```
y <- y * 2
plot(x, y); lines(x, y)
```

- y * 2 multiplies all the values of y by two
- Plots points and a line with a slope of 2
Datasets
- 100 famous datasets are built in
- Use the function `data()` to get a list
- Datasets are stored as *dataframes objects*
  - Objects made up of rows, columns, and names

The *InsectSprays* dataset
- View the contents of a dataframe
  - Type `head(InsectSprays) <Enter>`
- View summary information
  - Type `summary(InsectSprays) <Enter>`
Our First R Session

- The *InsectSprays* dataset
  - Attaching to an object
    - Type `attach(InsectSprays)`<Enter>
  - Plot a histogram
    - Type `hist(InsectSprays$count)`<Enter>
  - Graph a box-and-whisker plot
    - Type `boxplot(count~spray)`<Enter>
Our First R Session

- The *InsectSprays* dataset
  - Saving a dataframe to a file
    - Type `write.csv(InsectSprays, "~/insect.csv")<Enter>`
  - Loading a dataframe from a file
    - Use MS Excel to load `insect.csv` and save as tab delimited:
      - `C:\Documents and Settings\Users\Documents\insect.txt`
    - Type `insect <- read.delim("~/insect.txt")`
Your R objects are stored in a workspace

- To list the objects in your workspace (may be a lot):
  \[
  > \text{ls}()
  \]

- To save your workspace to a file:
  \[
  > \text{save.image}()
  \]

  The default workspace file is \~/.Rdata

To save single objects:
\[
> \text{save(file=“data.RData”,x,y)}
\]
Our First R Session

- To remove objects which you don’t need any more:
  \[ \texttt{rm(x, y)} \]

- To remove ALL objects in your workspace:
  \[ \texttt{rm(list=ls())} \]

- To load your workspace from a file:
  \[ \texttt{load(“data.RData“)} \]

- And now, what are we expecting to be present in the workspace?
Help and other resources

- Starting the R installation help pages
  > help.start()
- In general:
  > help("functionname")
- If you don’t know the function you’re looking for:
  > help.search("quantile")

- “What’s in this variable”?  
  > class(x)
  [1] “integer”

  > str(x)
  int [1:5] 1 2 3 4 5

  > summary(x)
  Min. 1st Qu. Median       Mean 3rd Qu.       Max.
  1      2      3           4          5

Basic data types
Containers that contain data

Types of objects: vector, factor, array, matrix, dataframe, list, function

Attributes
- mode: numeric, character (=string!), complex, logical
- length: number of elements in object

Creation
- assign a value
- create a blank object
Identifiers (object names)

- must start with a letter (A-Z or a-z)
- can contain letters, digits (0-9), periods (".")
  - Periods have no special meaning (i.e., unlike C or Java!)
- case-sensitive:
  e.g., mydata different from MyData
Assignments

- "<-" used to indicate assignment

```r
x <- 4711
x <- "hello world!"
x <- c(1,2,3,4,5,6,7)
x <- c(1:7,8)
assign("x", c(seq(1,7),8))
x <- 1:8
```

- *note: as of version 1.4 "=" is also a valid assignment operator*
- *You can also use assign("x", 4711)*
Basic (atomic) data types

- **Logical**
  ```r
  > x <- T; y <- F
  > x; y
  [1] TRUE
  [1] FALSE
  ```

- **Numerical**
  ```r
  > a <- 5; b <- sqrt(2)
  > a; b
  [1] 5
  [1] 1.414214
  ```

- **Strings** *(called “characters”!)*
  ```r
  > a <- "1"; b <- 1
  > a; b
  [1] "1"
  [1] 1
  > a <- "string"
  > b <- "a"; c <- a
  > a; b; c
  [1] "string"
  [1] "a"
  [1] "string"
  ```
But there is more!

R can handle “big chunks of numbers” in elegant ways:

- **Vector**
  - Ordered collection of data of the same data type
  - Example:
    - Download timestamps
    - Last names of all students in this class
  - In R, a single number is a vector of length 1

- **Matrix**
  - Rectangular table of data of the same data type
  - Example: a table with marks for each student for each exercise

- **Array**
  - Higher dimensional matrix of data of the same data type
  - (Lists, data frames, factors, function objects, ... → later)
Vectors

```r
> Mydata <- c(2, 3.5, -0.2)  # Vector (c="concatenate")

> colours <- c("Black", "Red", "Yellow")  # String vector

> x1 <- 25:30
> x1
[1] 25 26 27 28 29 30  # Number sequence

> colours[1]  # Index starts with 1, not with 0!!!
[1] "Black"  # Addressing one element…

> x1[3:5]
[1] 27 28 29  # …and multiple elements
```
More examples with vectors:

```r
>x <- c(5.2, 1.7, 6.3)
>log(x)
 [1] 1.6486586 0.5306283 1.8405496
>y <- 1:5
>z <- seq(1, 1.4, by = 0.1)
>y + z
 [1] 2.0 3.1 4.2 5.3 6.4
>length(y)
[1] 5
>mean(y + z)
[1] 4.2
```
Subsetting

- Often necessary to extract a subset of a vector or matrix
- R offers a couple of neat ways to do that:

```r
x <- c("a", "b", "c", "d", "e", "f", "g", "a")
> x[1] # first (!) element
> x[3:5] # elements 3..5
> x[-(3:5)] # elements 1 and 2
> x[c(T,F,T,F,T,F,T,F)] # even-index elements
> x[x<= "d"] # el. "a"..."d","a"
```
Typical operations on vector elements

> Mydata
[1] 2 3.5 -0.2
> Mydata > 0
[1] TRUE TRUE FALSE

> Mydata[Mydata > 0]
[1] 2 3.5

> Mydata[-c(1,3)]
[1] 3.5

> which(Mydata > 0)
[1] 1 2

- Test on the elements
- Extract the positive elements
- Remove the given elements
> x <- c(5, -2, 3, -7)
> y <- c(1, 2, 3, 4) * 10

Multiplication on all the elements

> y
[1] 10 20 30 40

> sort(x)

Sorting a vector
[1] -7 -2 3 5

> order(x)

[1] 4 2 3 1

Element order for sorting

> y[order(x)]
[1] 40 20 30 10

Operation on all the components

> rev(x)

Reverse a vector
[1] -7 3 -2 5
Matrix: Rectangular table of data of the same type

```r
> m <- matrix(1:12, 4, byrow = T); m
> y <- -1:2
> m.new <- m + y
> t(m.new)
> dim(m)
> dim(t(m.new))
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
</tbody>
</table>

4 x 3

3 x 4
Matrix: Rectangular table of data of the same type

```r
>x <- c(3, -1, 2, 0, -3, 6)
>x.mat <- matrix(x, ncol=2)    # Matrix with 2 cols
>x.mat
[,1] [,2]
[1,]  3  0
[2,] -1 -3
[3,]  2  6

>x.matB <- matrix(x, ncol=2, byrow=T) # By-row creation
>x.matB
  [,1] [,2]
[1,]  3 -1
[2,]  2  0
[3,] -3  6
```
Building subvectors and submatrices

```r
> x.matB[,2]  # 2nd column
[1] -1 0 6

> x.matB[c(1,3),]  # 1st and 3rd lines
 [,1] [,2]
[1,]  3  -1
[2,] -3   6

> x.mat[-2,]  # Everything but the 2nd line
 [,1] [,2]
[1,]  3  0
[2,]  2  6
```
Dealing with matrices

```r
> dim(x.mat)  # Dimension (i.e., size)
[,1] 3 2
> t(x.mat)   # Transposition
[,1] [,2] [,3]
[1,] 3 2 -3
[2,] -1 0  6

> x.mat %*% t(x.mat)  # Matrix multiplication; also see %o%
[,1] [,2] [,3]
[1,] 10  6 -15
[2,]  6  4 - 6
[3,] -15 -6 45

> solve()  # Inverse of a square matrix
> eigen()  # Eigenvectors and eigenvalues
```
R is designed to handle statistical data
=> Has to deal with missing / undefined / special values
Multiple ways of missing values
  ▪ NA: not available
  ▪ NaN: not a number
  ▪ Inf, -Inf: infinity
Different from Perl: NaN ≠ Inf ≠ NA ≠ FALSE ≠ ""
  ≠ 0 (pairwise)
NA also may appear as Boolean value
i.e., boolean value in R ∈ {TRUE, FALSE, NA}
Special values (2/3)

- **NA**: Numbers that are “not available"
  
  ```r
  > x <- c(1, 2, 3, NA)
  > x + 3
  [1] 4  5  6 NA
  ```

- **NaN**: “Not a number"
  
  ```r
  > 0/0
  [1] NaN
  ```

- **Inf, -Inf**: infinite
  
  ```r
  > log(0)
  [1] -Inf
  ```
Odd (but logical) interactions with equality tests, etc:

```r
> 3 == 3
[1] TRUE
> 3 == NA
[1] NA
> NA == NA
[1] NA
> NaN == NaN
[1] NA
> 99999 >= Inf
[1] FALSE
> Inf == Inf
[1] TRUE
```

#but not “TRUE”!

And what about

```r
> 1+1+1 == 3
[1] TRUE
> 0.1 + 0.1 + 0.1 == 0.3
[1] FALSE
```
Lists
**Lists (1/4)**

**vector:** an ordered collection of data **of the same type.**

```r
> a = c(7,5,1)
> a[2]
[1] 5
```

**list:** an ordered collection of data **of arbitrary types.**

```r
> doe = list(name="john",age=28,married=F)
> doe$name
[1] "john"
> doe$age
[1] 28
```

Typically, vector/matrix elements are accessed by their index (=an integer), list elements by their name (=a string). **But both types support both access methods.**
A list is an object consisting of objects called *components*.

Components of a list *don’t need* to be of the same mode or type:

- \( \text{list1} \leftarrow \text{list}(1, 2, \text{TRUE}, \text{“a string”, 17}) \)
- \( \text{list2} \leftarrow \text{list}(\text{l1}, 23, \text{l1}) \)  # lists within lists: possible

A component of a list can be referred either as

- \( \text{listname}[[\text{index}]] \)

Or as:

- \( \text{listname}\$\text{componentname} \)
The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely.

Syntactic quicksand:
- aa[[1]] is the first component of aa
- aa[1] is the sublist consisting of the first component of aa only.

There are functions whose return value is a list (and not a vector / matrix / array)
Lists are very flexible

```r
> my.list <- list(c(5,4,-1), c("X1","X2","X3"))
> my.list
[[1]]:
[1]  5  4 -1

[[2]]:
[1] "X1" "X2" "X3"

> my.list[[1]]
[1]  5  4 -1

> my.list <- list(component1 = c(5,4,-1), component2 = c("X1","X2","X3"))
> my.list$component2[2:3]
[1] "X2" "X3"
```
> Empl <- list(employee = "Anna", spouse = "Fred", children = 3, child.ages = c(3, 7, 9))
> Empl[[1]]  # You'd achieve the same with: Empl$employee
  "Anna"
> Empl[[4]][2]
  7  # You'd achieve the same with: Empl$child.ages[2]
> Empl$child.a
[1] 3 7 9  # You can shortcut child.ages as child.a
> Empl[4]
$child.ages
[1] 3 7 9  # a sublist consisting of the 4th component of Empl
> names(Empl)
[1] "employee"  "spouse"  "children"  "child.ages"
> unlist(Empl)  # converts it to a vector. Mixed types will be converted to strings, giving a string vector.
Back to matrices: Naming elements of a matrix

```r
> x.mat
   [,1] [,2]
[1,]  3  -1
[2,]  2   0
[3,] -3   6

> dimnames(x.mat) <- list(c("Line1","Line2","xyz"),
                         c("col1","col2"))
#assign names to rows/columns of matrix

> x.mat
   col1 col2
Line1   3  -1
Line2   2   0
xyz    -3   6
```
Even more datatypes:
Data frames and factors
Data Frames (1/2)

- Vector: All components must be of same type
- List: Components may have different types
- Matrix: All components must be of same type

=> Is there an equivalent to a List?

- **Data frame:**
  - Data within each column must be of same type, but
  - Different columns may have different types (e.g., numbers, boolean, ...)
  - Like a spreadsheet

Example:
```r
> cw <- chickwts
> cw
weight    feed
11      309  linseed
23      243  soybean
37      423  sunflower
...```
Data Frames (2/2)

- Data frame = special list with class “data.frame”.
  - But: restrictions on lists that may be made into data frames.
- Components must be
  - vectors (numeric, character, or logical)
  - Factors
  - numeric matrices
  - Lists
  - other data frames.
- Matrices, lists, and data frames provide as many variables to the new data frame as they have columns, elements, or variables, respectively.
- Numeric vectors and factors are included as-is
- Non-numeric vectors are coerced to be factors, whose levels are the unique values appearing in the vector.
- Vector structures appearing as variables of the data frame must all have the same length, and matrix structures must all have the same row size.
Individual elements of a vector, matrix, array or data frame are accessed with “[]” by specifying their index, or their name

\[
\text{cw} = \text{chickwts}
\]

\[
\text{cw}
\]

weight feed
1  179  horsebean
11  309  linseed
23  243  soybean
...

\[
\text{cw[3,2]}
\]

[1] horsebean

6 Levels: casein horsebean linseed ... sunflower

\[
\text{cw [3,]}
\]

weight feed
37  423  sunflower
Subsetting in data frames (2/2)

> an = Animals
> an

<table>
<thead>
<tr>
<th>Species</th>
<th>Body</th>
<th>Brain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mountain beaver</td>
<td>1.350</td>
<td>8.1</td>
</tr>
<tr>
<td>Cow</td>
<td>465.000</td>
<td>423.0</td>
</tr>
<tr>
<td>Grey wolf</td>
<td>36.330</td>
<td>119.5</td>
</tr>
</tbody>
</table>

> an [3,]

<table>
<thead>
<tr>
<th>Species</th>
<th>Body</th>
<th>Brain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grey wolf</td>
<td>36.33</td>
<td>119.5</td>
</tr>
</tbody>
</table>
Labels in data frames

```r
labels <- c(
  "Mountainbeaver", "Cow",
  "Greywolf", "Goat",
  "Guineapig", "Dipliodocus",
  "Asianelephant", "Donkey",
  "Horse", "Potarmonkey",
  "Cat", "Giraffe",
  "Gorilla", "Human",
  "Africanelephant", "Triceratops",
  "Rhesusmonkey", "Kangaroo",
  "Goldenhamster", "Mouse",
  "Rabbit", "Sheep",
  "Jaguar", "Chimpanzee",
  "Horse", "Brachiosaurus",
  "Cat", "Pig",
  "Mole", "Rat",
  "Rhesusmonkey", "Mouse",
  "Goldenhamster", "Mouse",
  "Rabbit", "Sheep",
  "Jaguar", "Chimpanzee",
  "Horse", "Brachiosaurus",
  "Cat", "Pig",
  "Mole"
)
```

```
labels <- c(
  "body", "brain"
)
```
Factors

- A normal character string may contain arbitrary text
- A **factor** may only take pre-defined values
  - “Factor”: also called “category” or “enumerated type”
  - Similar to `enum` in C, C++ or Java 1.5
- `help(factor)`
Graphics in R
Web References

- [http://www.r-project.org](http://www.r-project.org)
  The official R home page

- [http://addictedtor.free.fr/graphiques](http://addictedtor.free.fr/graphiques)
  R graph gallery that shows off R’s graphing capability
**plot(): Scatterplots**

- A scatterplot is a standard two-dimensional (X,Y) plot
- Used to examine the relationship between two (continuous) variables

- If x and y are vectors, then
  - `plot(x,y)` produces a **scatterplot** of x against y
    - I.e., do a point at coordinates (x[1], y[1]), then (x[2], y[2]), etc.
  - `plot(y)` produces a **time series plot** if y is a numeric vector or time series object.
    - I.e., do a point a coordinates (1,y[1]), then (2, y[2]), etc.
- `plot()` takes lots of arguments to make it look fancier
  => `help(plot)`
The function `rnorm()` Simulates a random normal distribution.

Help `?rnorm`, `?runif`, `?rexp`, `?binom`, ...
Sometimes you don’t want just points

solution:

> plot(dataX, dataY, type="l")

Or, points and lines between them:

> plot(dataX, dataY, type="b")

Beware: If dataX is not nicely sorted, the lines will jump erroneously across the coordinate system

- try

  plot(rnorm(100,1,1), rnorm(100,1,1), type="l") and see what happens
Graphical Parameters of

plot(x, y, ...
  type = "c", #c may be p (default), l, b, s, o, h, n. Try it.
  pch="+", # point type. Use character or numbers 1 – 18
  lty=1, # line type (for type="l"). Use numbers.
  lwd=2, # line width (for type="l"). Use numbers.
  axes = "L" # L=F,T
  xlab="string", ylab="string" # Labels on axes
  sub = "string", main ="string" # Subtitle for plot
  xlim = c(lo, hi), ylim= c(lo, hi) # Ranges for axes
)

And some more.
Try it out, play around, read help(plot)
More example graphics with

```r
>x <- seq(-2*pi, 2*pi, length=100)
>y <- sin(x)

>par(mfrow=c(2,2))  # multi-plot
>plot(x, y, xlab="x", ylab="Sin x")

>plot(x, y, type="l", main="A Line")

>plot(x[seq(5, 100, by=5)],
     y[seq(5, 100, by=5)],
     type="b", axes=F)

>plot(x, y, type="n",
     ylim=c(-2, 1)
>par(mfrow=c(1,1))
```
Multiple data in one plot

- **Scatter plot**
  1. `> plot(firstdataX, firstdataY, col="red", pty="1", ...)`
  2. `> points(seconddataX, seconddataY, col="blue", pty="2")`
  3. `> points(thirddataX, thirddataY, col="green", pty=3)`

- **Line plot**
  1. `> plot(firstdataX, firstdataY, col="red", lty="1", ...)`
  2. `> lines(seconddataX, seconddataY, col="blue", lty="2", ...)`

- **Caution:**
  - Only `plot()` command sets limits for axes!
  - Avoid using `plot( ..... , xlim=c(bla, blubb), ylim=c (laber, rhabarber))`
  - (There are other ways to achieve this)
Logarithmic scaling

- `plot()` can do logarithmic scaling
  - `plot(.... , log="x")`
  - `plot(.... , log="y")`
  - `plot(.... , log="xy")`
- Double-log scaling can help you to see more. Try:
  ```r
  > x <- 1:10
  > x.rand <- 1.2^x + rexp(10,1)
  > y <- 10*(21:30)
  > y.rand <- 1.15^y + rexp(10, 20000)
  > plot(x.rand, y.rand)
  > plot(x.rand, y.rand, log="xy")
  ```
More nicing up your graph

> axis(1, at=c(2,4,5),
    legend("A","B","C"))
inside

> abline(lsfit(x,y))
Add an adjustment

> abline(0,1)
add a line of slope 1 and intercept 0

> legend(locator(1),...)
Legends: very flexible

Axis details ("ticks", legend, …)
Use xaxt="n" or yaxt="n"
plot()
A histogram is a special kind of bar plot.

It allows you to visualize the distribution of values for a numerical variable. Naïvely:

- Divide range of measurement values into, say, 10 so-called “bins”
- Put all values from, say, 1-10 into bin 1, from 11-20 into bin 2, etc.
- Count: how many values in bin 1? In bin 2? ...
- Then draw these counters

When drawn with a density scale:

- the **AREA** (NOT height) of each bar is the proportion of observations in the interval
- the **TOTAL AREA** is 100% (or 1)
Type `?hist` to view the help file

- Note some important arguments, esp `breaks`

Simulate some data, make histograms varying the number of bars (also called ‘bins’ or ‘cells’), e.g.

```r
>par(mfrow=c(2,2))  # set up multiple plots
>simdata<-rchisq(100,8)  # some random numbers
>hist(simdata)       # default nb of bins
>hist(simdata,breaks=2)  # etc,4,20
```
R: setting your own breakpoints

```r
> bps <- c(0,2,4,6,8,10,15,25)
> hist(simdata, breaks=bps)
```
Density plots

- Density: probability distribution
- Naïve view of density:
  - A “continuous”, “unbroken” histogram
  - “infinite number of bins”, a bin is “infinitesimally small”
  - Analogy: Histogram ~ sum, density ~ integral

- Calculate density and plot it
  ```r
  > x <- rnorm(200, 0, 1)  # create random numbers
  > plot(density(x))  # compare this to:
  > hist(x)
  ```
Other graphical functions

See also:

- `barplot()`
- `image()`
- `pairs()`
- `persp()`
- `piechart()`
- `polygon()`

- `library(modreg)`
- `scatter.smooth()`
Interactive Graphics Functions

- `locator(n,type="p")`: Waits for the user to select locations on the current plot using the left mouse button. This continues until `n` (default=500) points have been selected.

- `identify(x, y, labels)`: Allow the user to highlight any of the points defined by `x` and `y`.

- `text(x,y,"Hey")`: Write text at coordinate `x,y`.
Reading and writing files

- Different methods for input
  - Reading a vector (scan)
  - Reading a table (read.table, read.csv, ...)
  - File handles

- Different methods for output
  - Writing single strings
  - Writing tables into a file (write.table)
  - Saving plots as PostScript, PNG, ...
  - File handles
Simple input

- Task: Read a file into a vector
- Input file looks like this:
  
  1
  2
  17.5
  99
  99
- Read this into vector x:
  \[ x \leftarrow \text{scan("inputfile.txt")} \]
- There are more options \[=>\text{help(scan)} \]
Write a table into a file:

```r
> x <- rnorm(100, 1, 1)
> write.table(x, file="numbers.txt")
```

# There are more options => help(write.table)

Read a table from a file:

```r
> x <- read.table("in.txt", header=FALSE)
```

# There are more options => help(read.table)

Read a table from the Web:

```r
> x <- read.table("http://www.unipd.it/...")
```
Universal: Using file handles

- File handles about as universal as in Perl
- Write two lines into a file:
  ```r
  > fh <- file("output.txt", "w")  # write
  > cat("blah", "blubb", sep="\n", file=fh)
  > close(fh)
  ```
- Write into a file and compress it using gzip:
  ```r
  > fh <- gzfile("output.txt.gz", "w")
  > cat("blah blah blah blah", ..., file=fh)
  ```
- More examples: help(file)
- Also try “filenames” like http://www.blabla.bla/data.gz
Graphical output: Saving your plots

- Output as Portable Data Format:
  ```r
  > pdf("outputfile.pdf")
  > plot(data)  # You will not see this on screen!
  > ...  # do some more graphics
  > dev.off()  # write into file
  ```
  - There are many more options => help(postscript)
  - View the file using, e.g., `gv` program

- Output as PNG (bitmap):
  ```r
  Simply replace `postscript()` above by `png()`:
  > png("outputfile.png", width=800, height=600, pointsize=12, bg="white")
  ```
Useful built-in functions
Useful functions

```r
> seq(2,12,by=2)
[1]  2  4  6  8 10 12
> seq(4,5,length=5)
[1] 4.00 4.25 4.50 4.75 5.00
> rep(4,10)
[1] 4 4 4 4 4 4 4 4 4 4

> paste("V",1:5,sep=" ")
[1] "V1" "V2" "V3" "V4" "V5"

> LETTERS[1:7]
[1] "A" "B" "C" "D" "E" "F" "G"
```
Mathematical operations

Normal calculations: +  -  *  /  
Powers:  $2^5$ or as well  $2**5$
Integer division:  \( \%/\% \)
Modulus:  \( \%\% \)  \(7\%\%5\) gives 2

Standard functions:  \texttt{abs()}, \texttt{sign()}, \texttt{log()}, \texttt{log10()}, \texttt{sqrt()}, \texttt{exp()}, \texttt{sin()}, \texttt{cos()}, \texttt{tan()}

To round:  \texttt{round(x,3)}  rounds to 3 figures after the point

And also:  \texttt{floor(2.5)} gives 2,  \texttt{ceiling(2.5)} gives 3

All this works for matrices, vectors, arrays etc. as well!
```r
vec <- c(5, 4, 6, 11, 14, 19)
> sum(vec)
[1] 59
> prod(vec)
[1] 351120
> mean(vec)
[1] 9.833333
> var(vec)
[1] 34.96667
> sd(vec)
[1] 5.913262

And also: min()  max()
cummin()  cummax()
range()
```
**Logical functions**

R knows two logical values: **TRUE** (short T) et **FALSE** (short F). And **NA**.

Example:

```r
> 3 == 4
[1] FALSE
> 4 > 3
[1] TRUE
```

```r
> x <- -4:3
> x > 1
[1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
```

```r
> sum(x[x>1])
[1] 5
```

```r
> sum(x>1)
[1] 2
```

Notez la différence !
Programming: Control structures and functions
x = 1:9

if (length(x) <= 10) {
    x <- c(x, 10:20);  # append 10...20 to vector x
    print(x)
} else {
    print(x[1])
}
Loops in R

```r
list <- c(1,2,3,4,5,6,7,8,9,10)
for(i in list) {
  x[i] <- rnorm(1)
}

j = 1
while( j < 10) {
  print(j)
  j <- j + 2
}
```
Functions

- Functions do things with data
  - “Input”: function arguments (0, 1, 2, ...)
  - “Output”: function result (exactly one)

Example:
> pleaseadd <- function(a, b) {
+ result <- a + b
+ return(result)
+ }

Editing of functions:
> fix(pleaseadd)  # opens pleaseadd() in editor

Editor to be used determined by shell variable $EDITOR
Two ways of submitting parameters
- Arguments may be specified in the same order in which they occur in function definition
- Arguments may be specified as name=value. Here, the ordering is irrelevant.

Above two rules can be mixed!

```r
> t.test(x1, y1, var.equal=F, conf.level=.99)
> t.test(var.equal=F, conf.level=.99, x1, y1)
```
R function can handle missing arguments two ways:
- either by providing a default expression in the argument list of definition or
- by testing explicitly for missing arguments

```r
> add <- function(x,y=0){x + y}
> add(4)

> add <- function(x,y){
  if(missing(y)) x
  else x+y
}
> add(4)
```
The special argument name “...” in the function definition will match any number of arguments in the call.

nargs() returns the number of arguments in the current call.
> mean.of.all <- function(...) mean(c(...))
> mean.of.all(1:10,20:100,12:14)

> mean.of.means <- function(...) {
  means <- numeric()
  for(x in list(...)) means <- c(means,mean(x))
  mean(means)
}
mean.of.means <- function(...) {
  n <- nargs()
  means <- numeric(n)
  all.x <- list(...)
  for(j in 1:n) means[j] <- mean(all.x[[j]])
  mean(means)
}
mean.of.means(1:10, 10:100)
Hash tables
Hash Tables

- In vectors, lists, dataframes, arrays:
  - elements stored one after another
  - accessed in that order by their index == integer
  - or by the name of their row / column
- Now think of Perl’s hash tables, or java.util.HashMap
- R has hash tables, too
In R, a hash table is the same as a workspace for variables, which is the same as an environment.

```r
> tab = new.env(hash=T)

> assign("btk", list(cloneid=682638, fullname="Bruton agammaglobulinemia tyrosine kinase"), env=tab)

> ls(env=tab)
[1] "btk"

> get("btk", env=tab)
$cloneid
[1] 682638
$fullname
[1] "Bruton agammaglobulinemia tyrosine kinase"
```
Object orientation
primitive (or: atomic) data types in R are:

- numeric (integer, double, complex)
- character
- logical
- function
- out of these, vectors, matrices, arrays, lists can be built
Object orientation

- Object: a collection of atomic variables and/or other objects that belong together
- Similar to the previous list examples, but there’s more to it.

- Parlance:
  - class: the “abstract” definition of it
  - object: a concrete instance
  - method: other word for ‘function’
  - slot: a component of an object (I.e., object variable)
Object orientation advantages

The usual suspects:

- Encapsulation (can use the objects and methods someone else has written without having to care about the internals)
- Generic functions (e.g. `plot`, `print`)
- Inheritance (hierarchical organization of complexity)
library('methods')
setClass('microarray',
   ## the class definition
   representation(  
      ## its slots
      qua = 'matrix',
      samples = 'character',
      probes = 'vector'),
   prototype = list(  
      ## and default values
      qua = matrix(nrow=0, ncol=0),
      samples = character(0),
      probes = character(0)))

dat = read.delim('../data/alizadeh/lc7b017rex.DAT')
z = cbind(dat$CH1I, dat$CH2I)

setMethod('plot',
   ## overload generic function 'plot'
   signature(x='microarray'),  
   ## for this new class
   function(x, ...)
   plot(x@qua, xlab=x@samples[1], ylab=x@samples[2], pch='.', log='xy'))

ma = new('microarray',
   ## instantiate (construct)
   qua = z,
   samples = c('brain','foot'))
plot(ma)
The `plot(pisa.linearmodel)` command is different from `plot(year,inclin)`.

`plot(pisa.linearmodel)`

R recognizes that `pisa.linearmodel` is a “lm” object.
Thus it uses `plot.lm()`.

Most R functions are object-oriented.

For more details see `?methods` and `?class`