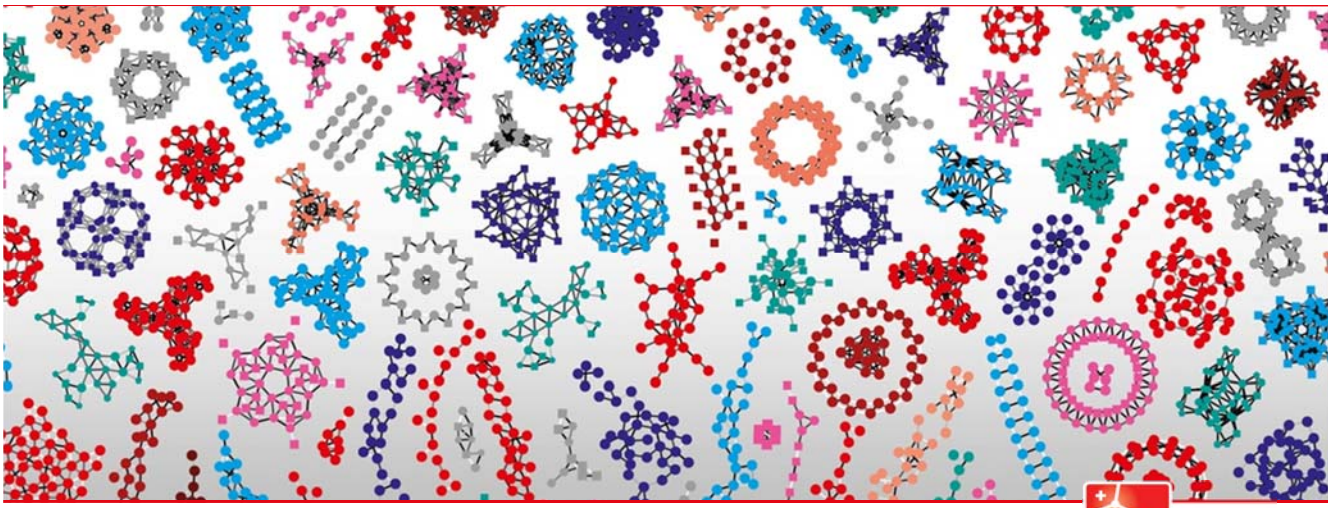
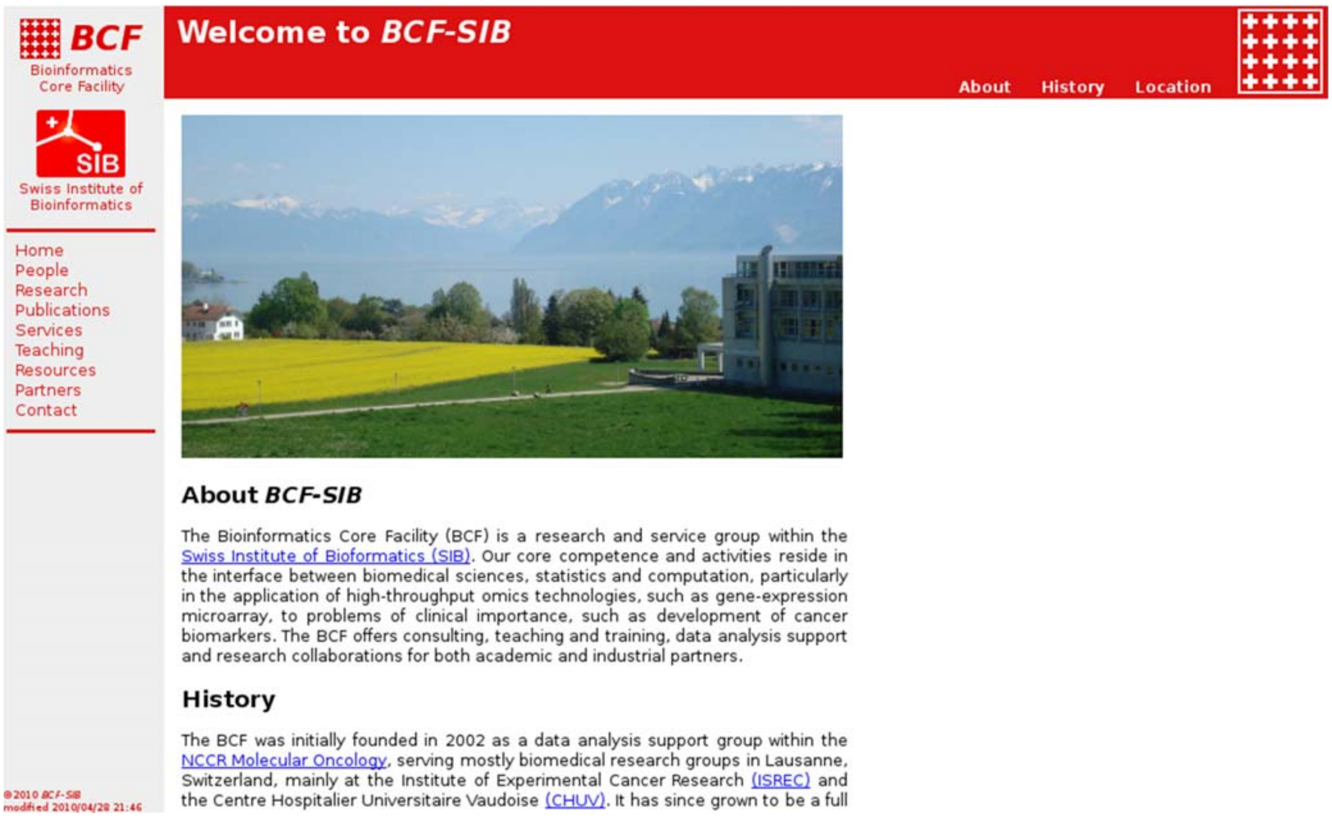


Swiss Institute of Bioinformatics

Advanced R
December 2017

Frédéric Schütz (Frederic.Schutz@sib.swiss)

www.sib.swiss



BCF
Bioinformatics Core Facility

Swiss Institute of Bioinformatics

Home
People
Research
Publications
Services
Teaching
Resources
Partners
Contact

©2010 BCF-SIB
modified 2010/04/28 21:46

Welcome to *BCF-SIB*

About History Location

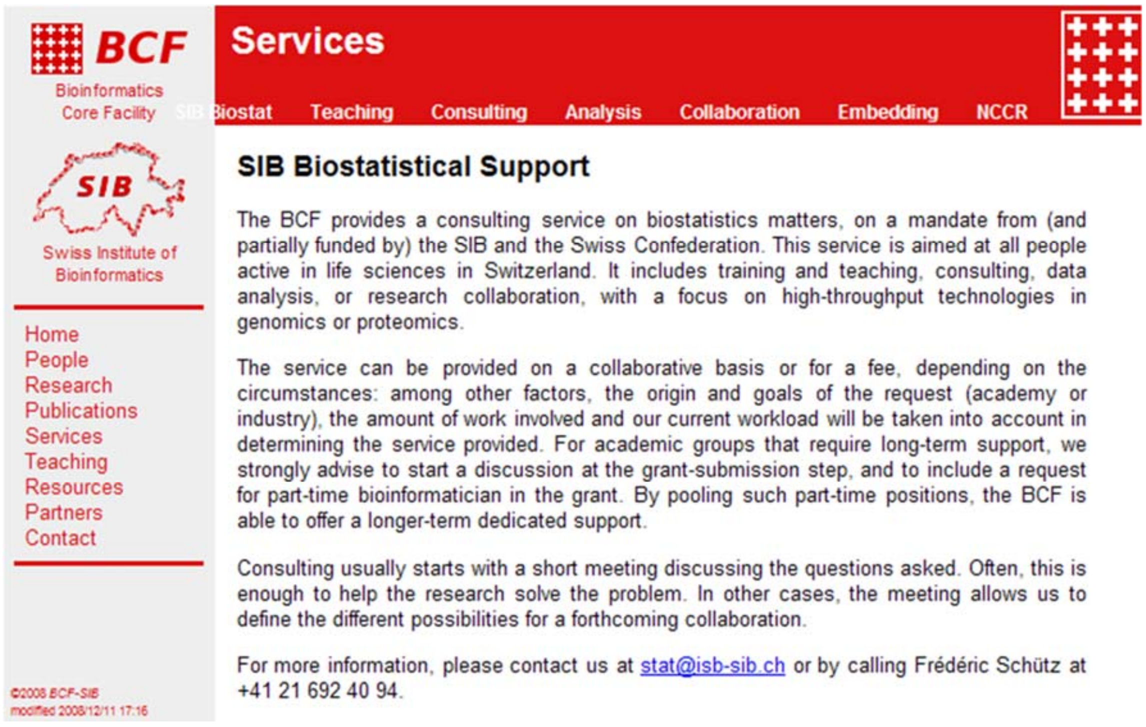
About *BCF-SIB*

The Bioinformatics Core Facility (BCF) is a research and service group within the Swiss Institute of Bioformatics (SIB). Our core competence and activities reside in the interface between biomedical sciences, statistics and computation, particularly in the application of high-throughput omics technologies, such as gene-expression microarray, to problems of clinical importance, such as development of cancer biomarkers. The BCF offers consulting, teaching and training, data analysis support and research collaborations for both academic and industrial partners.

History

The BCF was initially founded in 2002 as a data analysis support group within the NCCR Molecular Oncology, serving mostly biomedical research groups in Lausanne, Switzerland, mainly at the Institute of Experimental Cancer Research (ISREC) and the Centre Hospitalier Universitaire Vaudoise (CHUV). It has since grown to be a full

http://bcf.isb-sib.ch/



SIB Biostatistical Support

The BCF provides a consulting service on biostatistics matters, on a mandate from (and partially funded by) the SIB and the Swiss Confederation. This service is aimed at all people active in life sciences in Switzerland. It includes training and teaching, consulting, data analysis, or research collaboration, with a focus on high-throughput technologies in genomics or proteomics.

The service can be provided on a collaborative basis or for a fee, depending on the circumstances: among other factors, the origin and goals of the request (academy or industry), the amount of work involved and our current workload will be taken into account in determining the service provided. For academic groups that require long-term support, we strongly advise to start a discussion at the grant-submission step, and to include a request for part-time bioinformatician in the grant. By pooling such part-time positions, the BCF is able to offer a longer-term dedicated support.

Consulting usually starts with a short meeting discussing the questions asked. Often, this is enough to help the research solve the problem. In other cases, the meeting allows us to define the different possibilities for a forthcoming collaboration.

For more information, please contact us at stat@isb-sib.ch or by calling Frédéric Schütz at +41 21 692 40 94.

©2008 BCF-SIB
modified 2008/12/11 17:16

http://bcf.isb-sib.ch/Services.html

Reproducible research

"Research is reproducible if it can be reproduced by others"

Of course, rerunning an experiment will give different results—an observation that gave rise to the development of statistics as a discipline.

Our focus here is "reproducible research" (RR) in the sense of reproducing conclusions from a single experiment based on the measurements from that experiment.

Amstat News, 1 January 2011

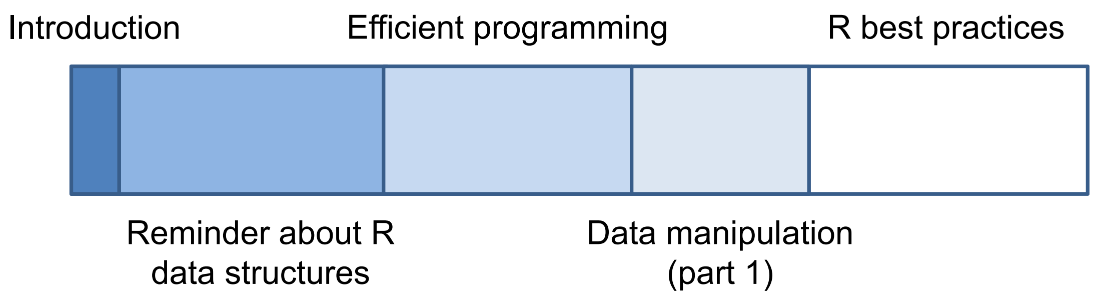
✓ Reproducible Research

Definition of reproducible research

A complete description of the data and the analysis of that data — including computer programs — so the results can be exactly reproduced by others.

Amstat News, 1 January 2011

Content



References

- Hadley Wickham. «Advanced R». CRC Press, 2014
- Phil Spector. «Data manipulation with R». Springer, 2008
- W. Venables and B. Ripley. «S Programming». Springer, 2004
- John M. Chambers. «Software for Data Analysis – Programming with R». Springer, 2008

```
c=c(c=c)

c=c(c="c")
```

An introduction (or reminder) about R data structures

What are the main objects in R ?

Vectors

The most important objects in R are vectors

- **Atomic vectors**: an ordered collection of data of the same type
- **Lists:** an ordered collection of data that can be of different types.

Attributes

Attributes are arbitrary labels attached to the R objects.

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
> attributes(x)
$mylabel
[1] "Random normal data"

> attr(x, "class") <- "randomdata"
```

Some important attributes in R

- **names**: allows naming of the component of an object
- **class:** a label attached to the object, which indicates how actions can be performed on the object
- **dim:** the dimensions of the objects (e.g. for a matrix or an array)

```
> names(x) <- LETTERS[1:10]
> x
          A           B           C           D           E           F 
-0.93205027 -0.16194958  0.26727310 -0.07427123  1.54048877 -0.63579513 
          G           H           I           J 
 0.27141749 -2.03039854 -2.52658864  1.02263626 
attr(,"mylabel")
[1] "Random normal data"
attr(,"class")
[1] "randomdata"

> attributes(x)
$mylabel
[1] "Random normal data"

$class
[1] "randomdata"

$names
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Types and modes

All objects in R have a **type**, which describes the type of data stored in the object.

Sometimes, we also talk about the **mode**, a simplified version of types.

Type / Mode

The type can be (see `typeof(object)`) :

– **logical**	
– **integer**	(numeric)
– **double**	(numeric)
– **closure**	(function)
– **builtin**	(function)
– **special**	(function)
– **complex**	
– **character**	
– **raw**	
– **list**	
– (and a few others)	

(in parenthesis: mode, as indicated by the `mode()` function)

Type / Mode

The type can be (see `typeof(object)`) :

– **logical**	
– **integer**	(numeric)
– **double**	(numeric)
– **closure**	(function)

```
> f <- function() {}
> f$a
Error in f$a : object of type 'closure' is not subsettable
```

Logical vs numeric

Logical values (TRUE/FALSE) are very easy to convert to numeric value (0/1) and back, as in most programming languages:

```
> as.numeric( c(FALSE, TRUE) )
[1] 0 1
> as.logical( c(0,1) )
[1] FALSE  TRUE
> c(FALSE, 0)
[1] 0 0
> c(FALSE, 0, TRUE)
[1] 0 0 1
```

Logical vs numeric

This is very useful, for example for counting purposes.

In the example below: count how many elements of the vector `data` are larger than zero:

```
> data <- rnorm(10)
> data
 [1] -0.61518461 -0.62574053  1.21586046 -1.42627945
 [5]  0.06749257  0.59811401  0.25876230 -0.45936110
 [9] -1.83171441  0.28693148
> data > 0
 [1] FALSE FALSE  TRUE FALSE  TRUE
 [6]  TRUE  TRUE FALSE FALSE  TRUE
> sum(data > 0)
[1] 5
> mean(data > 0)
[1] 0.5
```

Difference between logical and numeric

However, in contrast to other programming languages, they can not be freely exchanged:

```
> vector <- 1:10

> vector[ c(0,1) ]
[1] 1
> vector[ c(F,T) ]
[1]  2  4  6  8 10
```

Difference between logical and numeric

```
> vector <- 1:10

> vector[ c(0,1) ]
[1] 1
```

This selects elements 0 (which does not exist) and 1 (=1)

```
> vector[ c(F,T) ]
[1]  2  4  6  8 10
```

This applies to each element in turn; since the logical vector is not long enough, it is recycled to cover the full vector. At the end, only elements at even positions are selected.

Storing data into R

The simplest way to store data into R is the vector, which contains an ordered collection of objects **of the same type**:

```
> x <- c(1,2,3,4); x
[1] 1 2 3 4
> typeof(x); mode(x)
[1] "double"
[1] "numeric"

> x <- c(1,2,TRUE,3); x
[1] 1 2 1 3
> typeof(x)
[1] "double"

> x <- c(1,2,"true",4); x
[1] "1"    "2"    "true" "4"
> typeof(x)
[1] "character"
```

Arrays and matrices

Matrices (in 2D) and arrays (in 2D or more) are an extension of vectors, where two or more dimensions are specified.

```
> m <- matrix(1:30, ncol=6)

> m[11]; m[1,3]              # Equivalent
[1] 11
[1] 11
> dim(m)
[1] 5 6
> length(m)
[1] 30
```

Arrays

Arrays are constructed in a similar way.

```
> a <- 1:24
> array(a, dim=c(4,3,2))
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

A matrix is a vector

In fact, a matrix (or array) is stored as a vector (column by column) with additional information about its dimensions.

```
> a <- 1:30
> attr(a, "dim") <- c(5,6)
> class(a) <- "matrix"
> a
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
```

Creating a matrix row by row

A matrix can also be created row by row, using the `byrow` parameter.

However, it will still be stored column by column.

```
> m <- matrix(1:30, ncol=6, byrow=TRUE); m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    7    8    9   10   11   12
[3,]   13   14   15   16   17   18
[4,]   19   20   21   22   23   24
[5,]   25   26   27   28   29   30

> as.vector(m)
  [1]  1  7 13 19 25  2  8 14 20 26  3  9 15 21 27
 [16]  4 10 16 22 28  5 11 17 23 29  6 12 18 24 30
```

Type of elements in a matrix

As for a vector, all elements of a matrix must be of the same type:

```
> typeof(a)
[1] "integer"
> a[3,3] <- "a"
> a
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "1"  "6"  "11" "16" "21" "26"
[2,] "2"  "7"  "12" "17" "22" "27"
[3,] "3"  "8"  "a"  "18" "23" "28"
[4,] "4"  "9"  "14" "19" "24" "29"
[5,] "5"  "10" "15" "20" "25" "30"
> typeof(a)
[1] "character"
```

Lists

Lists allow the storage of several objects (with different types) in a single R object.

```
> mylist <- list(ages=c(21, 32, 41, 45),
                 height=c(180, 176, 156, 165),
                 sex=c("M", "M", "F", "M") )
> mylist
$ages
[1] 21 32 41 45

$height
[1] 180 176 156 165

$sex
[1] "M" "M" "F" "M"
> class(mylist); typeof(mylist)
[1] "list"
[1] "list"
```

Lists

The objects can be accessed either using their rank, or by their name.

`[x]`	returns part (one element) of the list
`[[x]]`	returns what is inside this element

```
> mylist[1]
$ages
[1] 21 32 41 45
> typeof(mylist[1])
[1] "list"

> mylist[[1]]
[1] 21 32 41 45
> typeof(mylist[[1]])
[1] "double"

> mylist$height
[1] 180 176 156 165
```

Data frames

Data frames are usually the preferred method for working with datasets that consists of several observations (rows) on several variables (columns).

Data frames are an «easier to use» version of lists (where all elements of the list have the same length), and a more flexible version of matrices: they allow columns of differents types, while still making them easy to access.

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
> class(data); typeof(data)
[1] "data.frame"
[1] "list"
```

Data frames vs lists

Lists and data frames are similar: to convert the former into the latter, one only needs to:

- change the class to `data.frame`
- give (unique) names to the rows by setting the `row.names` attribute

```
> class(mylist) <- "data.frame"
> mylist
[1] ages   height sex
<0 rows> (or 0-length row.names)
> row.names(mylist) <- 1:length(mylist[[1]])
> mylist
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
```

Data frames

```
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M

> data[1]
  ages
1   21
2   32
3   41
4   45

> data[[1]]
[1] 21 32 41 45
```

Data frames: accessing columns by names

```
> data$height
[1] 180 176 156 165

> data[, "height"]
[1] 180 176 156 165

> data$h
[1] 180 176 156 165
```

When accessing a column by name, you can shorten the name as long as there is no ambiguity – although this is not recommended (the code may break if your script is used on a dataset that includes a new columns which causes an ambiguity).

The summary() command gives some brief information about an R object; its output depends on the type of object:

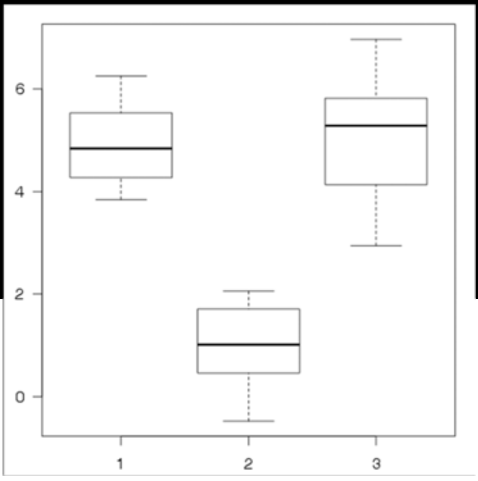
```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

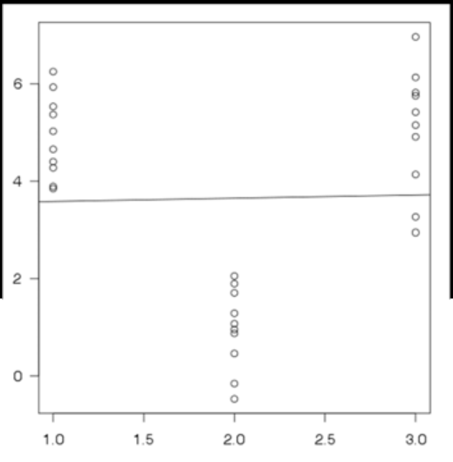
The str() command gives detailed information about the **str**ucture of an R object:

```
> str(mylist)
List of 3
 $ ages  : num [1:4] 21 32 41 45
 $ height: num [1:4] 180 176 156 165
 $ sex   : chr [1:4] "M" "M" "F" "M"

# Try this one if you don't believe the word "detailed" above
> model <- lm( runif(10) ~ rnorm(10) )
> str(model)
```

```
# Simulate data for 3 groups
set.seed(1)
groups <- rep( 1:3, each=10 )

measure <- vector(length=30)
measure[ groups==1 ] <- 5
measure[ groups==2 ] <- 1
measure[ groups==3 ] <- 5
measure <- measure + rnorm(30)

# Perform a one-way ANOVA on this data
boxplot( measure ~ groups )
summary(aov( measure ~ groups ) )
```

```
# Perform a one-way ANOVA on this data
> boxplot( measure ~ groups )
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```





Factors

Factors represent **categorical variables** in R.

They are vectors that can contain only values from a (finite) predefined set.

Example

```
> hair <- factor(c("blond", "brown", "red", "blond"))

> hair
[1] blond brown red   blond
Levels: blond brown red

> hair[2] <- "blond"
> hair[2] <- "grey"
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = "grey") :
  invalid factor level, NAs generated
> hair
[1] blond <NA>  red   blond
Levels: blond brown red
```

```
> class(hair)
[1] "factor"
> typeof(hair); mode(hair)
[1] "integer"
[1] "numeric"

> as.numeric(hair)
[1]  1 NA  3  1
> as.character(hair)
[1] "blond" NA      "red"   "blond"
```

Internally, R stores factors as integer numbers, along with the correspondance between number and labels (1=blond, 2=brown, 3=red).

Concatenating factors

```
> c(hair, hair)
[1] 1 2 3 1 1 2 3 1

# Workaround #1
> factor( as.character(hair), as.character(hair2))

# Workaround #2
> unlist( list( hair, hair) )
```

Simply concatenating factors will create a vector made out of the numeric values, which is almost certainly not what you want.

Ordered factors

Use the `ordered=TRUE` option for ordinal (ordered) values:

```
> time <- factor(c(1,2,3,2,2,1), levels=c(1,2,3),
               labels=c("never", "sometimes", "always"),
               ordered=TRUE)
> time
[1] never     sometimes always    sometimes
[5] sometimes never
Levels: never < sometimes < always
```

Some R functions respect ordered factors

Comparisons work as expected:

```
> time
[1] never     sometimes always    sometimes
[5] sometimes never
Levels: never < sometimes < always
> time[2] < time[3]
[1] TRUE
> "sometimes" < "always"
[1] FALSE
```

Some statistical modelling or plotting functions can adapt their parameters for ordered factors.

```
# Perform a one-way ANOVA on this data
> boxplot( measure ~ groups )
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816

> groups <- as.factor(groups)
> groups
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value   Pr(>F)
groups       2  94.12   47.06   52.95 4.53e-10 ***
Residuals   27  24.00    0.89
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

By default, `data.frame()` and `read.table()` convert all non-numerical values into factors.

This can be useful, or (more often…) it can be annoying.

Options to change this behaviour:

– `stringsAsFactors=FALSE`, or
– `as.is=TRUE` (for `read.table` only)

It can also be set by default using

```
options(stringsAsFactors=FALSE)
```

But this is not recommended, as your code may not work anymore if someone else uses it without specifying the same default option.

Factors and memory size

In previous versions of R, using factors for long vectors could save memory :

```
> f1 <- sample( c("Homo Sapiens", "Mus Musculus"), 10000,
              replace=TRUE)
> summary(f1)
   Length     Class      Mode
    10000 character character
> table(f1)
f1
Homo Sapiens         Mus Musculus
        4945                 5055
> f2 <- factor(f1)
> object.size(f1)
80168 bytes
> object.size(f2)
40544 bytes
```

Factors and memory size

In recent versions of R (2.6+) it is not the case anymore, as R stores only once each occurrence of a string in a vector:

```
> f1 <- sample( c("Homo Sapiens", "Mus Musculus"), 10000,
                replace=TRUE)
> summary(f1)
   Length     Class      Mode
    10000 character character
> table(f1)
f1
Homo Sapiens         Mus Musculus
        4945                 5055
> f2 <- factor(f1)
> object.size(f1)
40104 bytes
> object.size(f2)
40312 bytes
```

What we are not going to talk about…

Reading data into R

- `read.table()`, `scan()`, `read.csv()`, etc…

  did you know that these functions can directly access URLs ?

```
data <- read.table(
        "http://lausanne.isb-sib.ch/~schutz/data/class.txt")
```

- Reading zip, gzip or other compressed files
- Access other files (e.g. Excel files)
- Read/write to SQL databases

Reminder: getting information about R objects

The `summary()` command gives some brief information about an R object; its output depends on the type of object:

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```

Reminder: getting information about R objects

How does "`summary()`" know what to print for different objects ?

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```

Object-oriented programming in R

Fundamentals of object-oriented programming

Object: mechanism (usually data structure) that stores data and provides controlled access to it

Class: specification of the data and access mechanisms that a specific type of object supplies (blueprint)

Attribute: a piece of data owned by an object (or by a class)

Method: subroutine that provides some kind of access to an object's (or class's) data

Inheritance: reuse of attribute and method specifications from an existing class

Polymorphism: redefinition of behaviour of inherited methods

Adapted from Damian Conway, «*Introductory Object-Oriented Perl*»

Two frameworks for Object-oriented programming in R

S3 («old-style»)

- Informal, exists since the beginning
- Widely used, in particular in the base packages

S4 («formal classes»)

- More formal and rigorous, but less interactive
- Since R 1.7
- Used systematically in some contexts, e.g. Bioconductor

The S3 system

Every object has a class label attached to it, either

- explicitely set (using the `class()` function)
- matrix or array
- integer
- or the same as the mode of the object (`mode()`)

Examples of classes

```
> a <- c(1,1,2,3); class(a)
[1] "numeric"

> M <- matrix(1:4, ncol=2); class(M)
[1] "matrix"

> model <- lm( y ~ x ); class(model)
[1] "lm"

> f <- factor(a); class(f)
[1] "factor"
```

Getting a `summary()` of each of these variables

```
> summary(a)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1.00    1.00    1.50    1.75    2.25    3.00 
> summary(M)
       V1              V2       
 Min.   :1.00   Min.   :3.00  
 1st Qu.:1.25   1st Qu.:3.25  
 Median :1.50   Median :3.50  
 Mean   :1.50   Mean   :3.50  
 3rd Qu.:1.75   3rd Qu.:3.75  
 Max.   :2.00   Max.   :4.00  
> summary(model)
Call:
lm(formula = y ~ x)
[…]

Residual standard error: 0.1297 on 8 degrees of freedom
Multiple R-squared: 0.997,   Adjusted R-squared: 0.9967 
F-statistic:  2695 on 1 and 8 DF,  p-value: 2.102e-11
> summary(f)
1 2 3 
2 1 1 
```

***Method dispatch**: How does R creates the right summary ?*

The `summary()` function is defined as a generic function:

```
> summary
function (object, ...)
UseMethod("summary")
<environment: namespace:base>
```

If object `sheldon` is of class `bazinga`, when calling `summary(sheldon)`, R will search for a function called `summary.bazinga`, and will call

```
summary.bazinga(sheldon)
```

If `summary.bazinga` does not exist, R will call `summary.default(sheldon)` .

- Previous slide: should be 2 slides

Method dispatch

```
> methods("summary")
 [1] summary.aov            summary.aovlist       summary.aspell*
 [4] summary.connection     summary.data.frame    summary.Date
 [7] summary.default        summary.ecdf*         summary.factor
[10] summary.glm            summary.infl          summary.lm
[13] summary.loess*         summary.manova        summary.matrix
[16] summary.mlm            summary.nls*          summary.packageStatus*
[19] summary.PDF_Dictionary* summary.PDF_Stream*  summary.POSIXct
[22] summary.POSIXlt        summary.ppr*          summary.prcomp*
[25] summary.princomp*      summary.srcfile       summary.srcref
[28] summary.stepfun        summary.stl*          summary.table
[31] summary.tukeysmooth*

   Non-visible functions are asterisked
```

Note: to see the body of a non-visible function in R:

```
getS3method("summary", "princomp")
getAnywhere("summary.princomp")
```

Method dispatch

```
> methods(class="lm")
 [1] add1.lm*           alias.lm*          anova.lm
 [4] case.names.lm*     confint.lm*        cooks.distance.lm*
 [7] deviance.lm*       dfbeta.lm*         dfbetas.lm*
[10] drop1.lm*          dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*         formula.lm*
[16] hatvalues.lm       influence.lm*      kappa.lm
[19] labels.lm*         logLik.lm*         model.frame.lm
[22] model.matrix.lm    nobs.lm*           plot.lm
[25] predict.lm         print.lm           proj.lm*
[28] qr.lm*             residuals.lm       rstandard.lm
[31] rstudent.lm        simulate.lm*       summary.lm
[34] variable.names.lm* vcov.lm*
```

- What about coef() ?

The `print()` *method*

```
> model

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
  -0.001372     2.014997

> class(model)
[1] "lm"
> print(model)   # Equivalent to print.lm(model)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
  -0.001372     2.014997

# See print.lm for the details of how this information is printed
```

How to create an S3 object ?
Example: the mygsea2 package

1) Create a list(*) containing all the attributes your object need

```
mygsea2 <- function(small.list, big.list) {
  …
  z <- list(ks.pos=res$resks[1],  ks.neg=res$resks[2],
            p.pos=res$resperm[1], p.neg=res$resperm[2])
  z$nperms     <- n.perm
  z$weights    <- weights
  z$small.list <- small.list
  z$big.list   <- big.list


  z
}
```

(*) Any R object could be used, but lists are almost always used

2) Label it with the correct class

```
mygsea2 <- function(small.list, big.list) {
  …
  z <- list(ks.pos=res$resks[1],  ks.neg=res$resks[2],
            p.pos=res$resperm[1], p.neg=res$resperm[2])
  z$nperms     <- n.perm
  z$weights    <- weights
  z$small.list <- small.list
  z$big.list   <- big.list

  class(z) <- "gsea"
  z
}
```

3) Create the methods the user of the class/object will need

```
print.gsea <- function(object) {

  if (! any( class(object)=="gsea"))
    stop("Error: object is not a gsea object.")

  cat("GSEA analysis (", object$nperms," perms.)\n\n", sep="")
  cat("Small list: ", length(object$small.list),"\n",
      "  Big list: ", length(object$big.list),"\n\n", sep="")

  coefs <- cbind( c(object$ks.pos, object$ks.neg),
                  c(object$p.pos, object$p.neg) )
  colnames(coefs) <- c("Ks stat","P-value")
  rownames(coefs) <- c("+", "-")

  printCoefmat(coefs, P.values=TRUE, has.Pvalue=TRUE)
}
```

- Show examples (list methods, show results before/after)

4) If needed, create a new generic method

```
reduce.gsea <- function(object) {

  if (! any( class(object)=="gsea"))
    stop("Error: object is not a gsea object.")

  # Do something with the object
  ...
}

reduce <- function(object) UseMethod("reduce")
```

Reproducible
Research

Shortcomings of this informal system

The user can easily access the attributes directly (although he/she should not !), as with any other R object:

```
> class(model)
[1] "lm"
> names(model)
 [1] "coefficients"  "residuals"     "effects"      "rank"
 [5] "fitted.values" "assign"        "qr"           "df.residual"
 [9] "xlevels"       "call"          "terms"        "model"

> coef(model)                 # Recommended way
 (Intercept)            x
-0.001371868  2.014997472
> model$coefficients          # Not recommended
 (Intercept)            x
-0.001371868  2.014997472
```

Shortcomings of this informal system

The user can easily modify an attribute or the class itself, and R will not complain, unless you call a method that does not work anymore.

```
> class(model)
[1] "lm"
> model$coefficients <- c(0,0)
> model

Call:
lm(formula = y ~ x)

Coefficients:
[1]  0  0

> a <- 1:10; class(a) <- "lm"
> summary(a)
Error: $ operator is invalid for atomic vectors
```

The S4 model

- The S4 model is based on the same ideas («method dispatch») than S3

- It is however implemented in a much formal and stricter way.

- It also allows for «multiple dispatch»

Defining a class

```
setClass("GSEA",
         representation( nperms="numeric", weights="numeric",
                         small.list="character",
                         big.list="character"),
         contains="genelist",
         validity=function(object) {
           length(object@weights)==length(object@big.list)
         }
         )
```

Properties of a class include:

- A **name**
- A **representation**: list of attributes (*slots*) that the object contains
- **Inheritance**
- A **prototype** that specifies default values
- A **validation** function
- etc (see `?setClass`)

Creating an object

```
> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("a", "b", "c"),
              big.list=LETTERS[1:10])

> gsea <- new("GSEA", nperms="a", weights=1:10,
              small.list=c("a", "b", "c"),
              big.list=LETTERS[1:10])
Error in validObject(.Object) :
  invalid class “GSEA” object: invalid object for slot "nperms"
in class "GSEA": got class "character", should be or extend
class "numeric"

> gsea <- new("GSEA", nperms=10000, weights=1:10,
              small.list=c("a", "b", "c"),
              big.list="a")
Error in validObject(.Object) : invalid class “GSEA” object:
FALSE
```

Displaying an object: default output

```
> gsea
An object of class "GSEA"
Slot "nperms":
[1] 10000

Slot "weights":
 [1]  1  2  3  4  5  6  7  8  9 10

Slot "small.list":
[1] "a" "b" "c"

Slot "big.list":
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

The «`show`» method (equivalent to «`print`» in S3)

```
setMethod("show", "GSEA",
          function(object) {
            cat("GSEA with", object@nperms,"permutations.\n")
          }
          )


> gsea
GSEA with 10000 permutations.
```

Slots

Attributes in S4 objects are stored in *slots*.

They are similar to the components of a list for a S3 object, but well separated:

```
> slotNames(gsea)
[1] "nperms"     "weights"    "small.list" "big.list"

> gsea@nperms
[1] 10000

> gsea$nperms
Error in gsea$nperms : $ operator not defined for this S4 class
```

Note that you can still access and modify an object's content directly using the slots and the @ operator (and bypass any validation !), as with S3 objects, but you really, really should not (please ?)

How to list available methods

```
> showMethods("show")
Function: show (package methods)
object="ANY"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
...
> showMethods( class="GSEA")
Function: initialize (package methods)
.Object="GSEA"
    (inherited from: .Object="ANY")

Function: show (package methods)
object="GSEA"
```

Which system should I use ?

1) «While in Rome, Do as the Romans Do»:
   e.g. If your code fits with Bioconductor, use S4

2) Use S4 is there is a strong technical reason for doing so
   e.g. if you want to use objects directly in C++ code

3) Generally, use S3 objects and methods.

4) In any case, avoid mixing S3 and S4

Adapted from Google's R Style Guide:
https://google.github.io/styleguide/Rguide.xml

How to access some information in an unknown object ?

1) Look at `class(object)` (works with S3 and S4)
2) Look at its documentation
3) Find if the object is S3 or S4:
   – `names(object)` (empty for an S4 object)
   – `isS4(object)` (TRUE for an S4 object)
4) Look at the methods available for the object:
   – `methods(class="class")` for an S3 object
   – `showMethods(class="class")` for an S4 object

   and check whether one does what you need
5) Otherwise, look at its attributes (S3, $) or slots (S4, @)
6) If needed, look at a method to see how it handles the attributes:
   – `method.class` for an S3 object
   – `getMethods( "method", "class")` for an S4 object

RC: another framework for object-oriented development in R

- Introduced in R 2.12.0
- See: `?ReferenceClasses`

For more information…

- Thomas Lumley. “Programmer's Niche: A Simple Class, in S3 and S4” in R News 4/1, 2004, p. 33-36 http://cran.r-project.org/doc/Rnews/Rnews_2004-1.pdf
- https://github.com/hadley/devtools/wiki

Is there any practical difference between these two loops ?

```
set.seed(1)
n <- 5000; m <- 5000
a <- matrix( runif(n*m), ncol=n)

# Loop 1
for (i in 1:nrow(a)) {
  for (j in 1:ncol(a)) {
    b <- a[i,j]
  }
}


# Loop 2
for (i in 1:ncol(a)) {
  for (j in 1:nrow(a)) {
    b <- a[j,i]
  }
}
```

Efficient programming in R

Why care ?

Techniques used in other languages are often inefficient in R

In particular, they tend not to scale when the size of data increases.

R itself is not the fastest possible language

Finding which method is efficient or not is far from obvious (in R or any programming language).

Measuring the time used by an expression (I)

Use the commands:

```
library(microbenchmark)
microbenchmark(expression1, expression2, ...)
```

which runs the expressions 100 times (by default) and returns a summary of the running time.

Which one is fastest ?

```
> set.seed(1); x <- runif(100)

> sqrt(x)
> x^0.5

> microbenchmark( sqrt(x), x^0.5 )
Unit: microseconds
    expr    min      lq     mean  median     uq    max neval cld
 sqrt(x)  1.314  1.3720  1.80951  1.4190  1.460 33.621   100   a
   x^0.5 13.105 13.1805 13.48578 13.2405 13.328 31.875   100   b
```

Note: The last column (cld for "compact letter display") is only displayed if the `multcomp` package is installed.
It provides ranks for the different times, allowing for ties.

Measuring the time used by an expression (II)

Another command:

`system.time(expression)`

which returns three numbers:

user:	the time used to execute the expression itself
system:	the time used by the system while executing the expression (e.g. time spent reading files)
elapsed:	the total time spent (the one we are usually interested in)

Comparing codes: version 1

```
n <- 100000
m <- 100

results <- NULL

for (i in 1:n) {
    result <- mean( runif( m ) )
    results <- c(results, result)
}
```

Comparing code: version 2

```
n <- 100000
m <- 100

results <- vector("numeric", n)

for (i in 1:n) {
    result <- mean( runif( m ) )
    results[i] <- result
}
```

Comparing the two versions

```
system.time(
for (i in 1:n) {
    result <- mean( runif( m ) )
    ...
} )

                                        user   system elapsed
 results <- c(results, result)        21.433    1.264  22.778
 results[i] <- result                  1.780    0.000   1.782
```

One possible improvement: removing a temporary variable

```
n <- 100000
m <- 100

results <- vector("numeric", n)

for (i in 1:n) {

    results[i] <- mean( runif( m ) )
}
```

Comparing the three versions

```
system.time(
for (i in 1:n) {
    result <- mean( runif( m ) )
} )

                                        user   system elapsed
 results <- c(results, result)        21.433    1.264  22.778
 results[i] <- result                  1.780    0.000   1.782
 results[i] <- mean( runif( m ) )      1.832    0.000   1.836
```

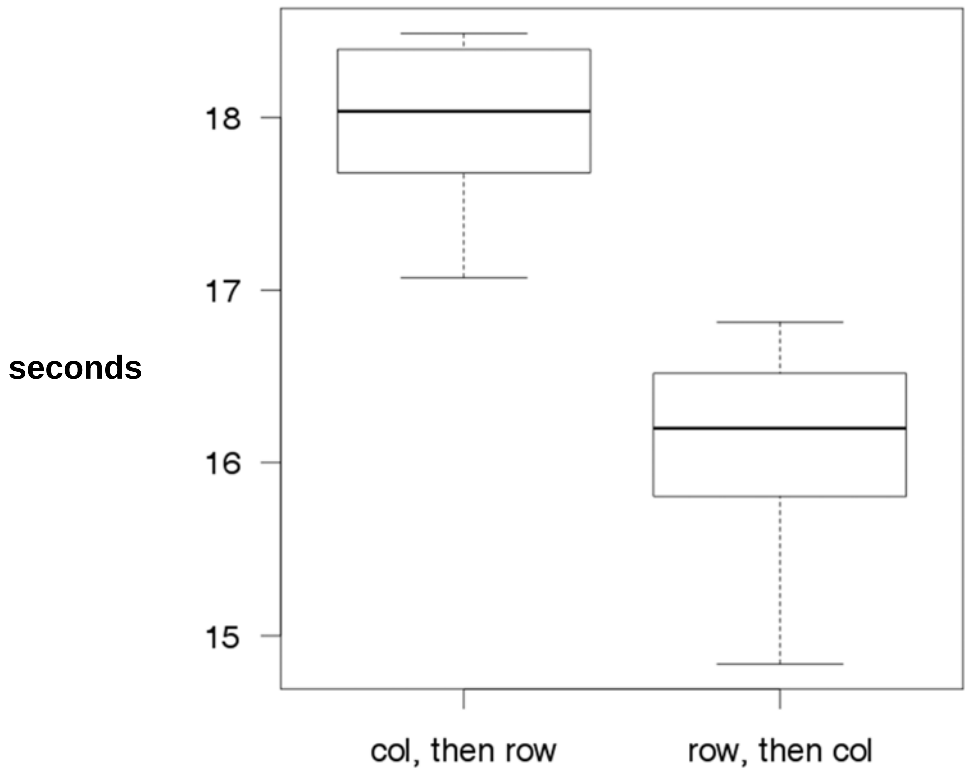
Is there any practical difference between these two loops ?

```
   set.seed(1)
   n <- 5000; m <- 5000
   a <- matrix( runif(n*m), ncol=n)

   system.time(
   for (i in 1:nrow(a)) {
     for (j in 1:ncol(a)) {
       b <- a[i,j]
     }
   }
   )

   system.time(
   for (i in 1:ncol(a)) {
     for (j in 1:nrow(a)) {
       b <- a[j,i]
     }
   }
   )
```

“The plural of anectodes is not data”

```
system.time(
for (i in 1:nrow(a)) {
  for (j in 1:ncol(a)) {
    b <- a[i,j]
  }
}
)
 user  system elapsed
 18.389   0.000  18.420

system.time(
for (i in 1:ncol(a)) {
  for (j in 1:nrow(a)) {
    b <- a[j,i]
  }
}
)
 user  system elapsed
 16.281   0.000  16.308
```

After repeating the test several times under different circumstances



Profiling

Profiling is a tool that allows the user to know how much time was spent on each part of his code.

It works by gathering information about what the code is doing at regular intervals (by default: every 20 ms, or 50 times per second) and saves it into the file.

Analyzing this file allows the user to find out which parts were slowest and may have to be rethought.

Example

```
Rprof()
pvalues <- NULL

for (i in 1:10000) {
  a <- runif(6)
  ttest <- t.test( a[1:3], a[4:6])
  pval <- ttest$p.value

  pvalues <- c(pvalues, pval)
}
Rprof(NULL)
```

Displaying the results of the profiling

```
summaryRprof()
> summaryRprof()
$by.self
                  self.time self.pct total.time total.pct
"deparse"              0.44    15.94       1.06     38.41
"t.test.default"       0.42    15.22       2.48     89.86
".deparseOpts"         0.24     8.70       0.30     10.87
"match"                0.20     7.25       0.64     23.19
"mean"                 0.18     6.52       0.24      8.70
"var"                  0.16     5.80       0.44     15.94
"stopifnot"            0.12     4.35       0.18      6.52
"pmatch"               0.12     4.35       0.12      4.35
"t.test"               0.10     3.62       2.60     94.20
"paste"                0.08     2.90       0.92     33.33
"mode"                 0.08     2.90       0.54     19.57
"c"                    0.08     2.90       0.08      2.90
"pt"                   0.08     2.90       0.08      2.90
"match.arg"            0.06     2.17       0.38     13.77
...
```

What we are not going to talk about…

Parallelizing code in R

- Markus Schmidberger, Martin Morgan, Dirk Eddelbuettel, Hao Yu, Luke Tierney, Ulrich Mansmann. “State of the Art in Parallel Computing with R”. Journal of Statistical Software 2009: JSS
- The CRAN Task View: High-Performance and Parallel Computing with R

Data manipulation/aggregation

Mapping a function to a matrix : `apply()`

```
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> apply(m, MAR=1, FUN=sum, na.rm=TRUE)
[1]  81  87  93  99 105
> rowSums(m)
[1]  81  87  93  99 105
```

`apply()` is generally faster than looping over all rows/columns. More specialized functions (e.g. `rowSums`) may be faster still.

Mapping a function to a matrix : `apply()`

```
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> apply(m, MAR=2, FUN=function(x) { c(mean(x), median(x) ) } )
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    3    8   13   18   23   28
[2,]    3    8   13   18   23   28
```

If the function returns more than one value for each row or column, `apply` will automatically create a matrix instead of a vector.

Mapping a function to a list : `lapply()`

```
> n <- as.list(as.data.frame(m)); n
$V1
[1] 1 2 3 4 5

$V2
[1]  6  7  8  9 10
...
> lapply(n, FUN=sum )
$V1
[1] 15

$V2
[1] 40
...

> sapply(n, FUN=sum )
 V1  V2  V3  V4  V5  V6 
 15  40  65  90 115 140
```

`lapply()` and `sapply()` both map a function to each element of a list; the first one returns a list, the other returns a vector or an array

How can we map a function to different groups ?

```
> head(data)
  sex height
1   M    183
2   M    183
3   M    182
4   M    175
5   M    158
6   M    179
```

Mapping a function to groups

```
> head(data)
  sex height
1   M    183
2   M    183
3   M    182
4   M    175
5   M    158
6   M    179
> tapply(data$height, data$sex, FUN=mean)
       F        M 
166.1739 178.2500 
```

Returns a vector or a list, depending on the output of the function (scalar or more complex object)

Mapping a function to groups given by several factors

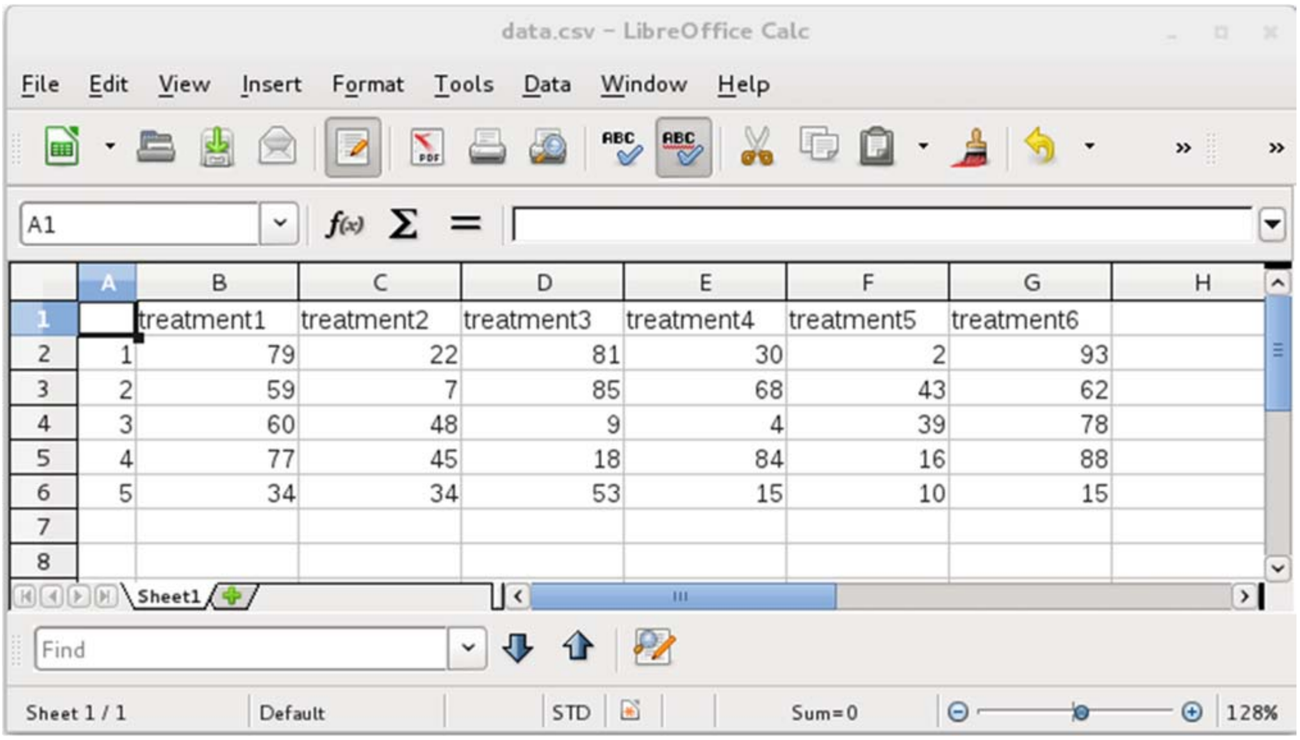
```
> head(data)
  sex height   smoking
1   M    183 nonsmoker
2   M    183 nonsmoker
3   M    182 nonsmoker
4   M    175 nonsmoker
5   M    158 nonsmoker
6   M    179    smoker
> tapply(data$height, list(data$sex, data$smoking), FUN=mean)
  nonsmoker smoker
F  166.3500    165
M  178.8421    176
```

Splitting data according to groups information

```
> split(data[,"taille"], data[,"sexe"])
$F
 [1] 172 165 165 156 172 168 166 176 159 164 150 163 169 160
[15] 165 170 173 165 159 175 170 168 172

$M
 [1] 183 183 182 175 158 179 185 177 186 178 183 178 183 177
[15] 180 174 184 168 169 181 170 180 184 181
```

Splits a vector (or rows of a data.frame) into separate elements of a list, ready for further processing.

A typical format for storing data



	A	B	C	D	E	F	G	H
1		treatment1	treatment2	treatment3	treatment4	treatment5	treatment6	
2	1	79	22	81	30	2	93	
3	2	59	7	85	68	43	62	
4	3	60	48	9	4	39	78	
5	4	77	45	18	84	16	88	
6	5	34	34	53	15	10	15	
7								
8								

Reproducible Research

What R needs to perform any modelling

```
> data
> values        ind
1       79 treatment1
2       59 treatment1
3       60 treatment1
4       77 treatment1
5       34 treatment1
6       22 treatment2
7        7 treatment2
8       48 treatment2
9       45 treatment2
...
> summary(aov( values ~ ind, data=data) )
```

How can we convert from one format to another ?

`stack` *and* `unstack`

```
> data
  treatment1 treatment2 treatment3 treatment4 treatment5 treatment6
1         79         22         81         30          2         93
2         59          7         85         68         43         62
3         60         48          9          4         39         78
4         77         45         18         84         16         88
5         34         34         53         15         10         15
> stack(data)
   values        ind
1      79 treatment1
2      59 treatment1
3      60 treatment1
4      77 treatment1
5      34 treatment1
6      22 treatment2
7       7 treatment2
...
```

For more complicated cases, the `reshape` function is efficient (but not easy to use !)

The `aggregate()` function

```
aggregate(  x,  by,  FUN  )
```

Data frame   List   Function

`aggregate()` works in a similar way to `tapply()`, but

- It works on whole data frames (multiple columns)
- It can only produce scalar summaries

The `aggregate()` function

```
> data(iris)
> head(iris, 3)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
> aggregate( iris[, 1:4], iris[5], FUN=mean )
     Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1     setosa        5.006       3.428        1.462       0.246
2 versicolor        5.936       2.770        4.260       1.326
3  virginica        6.588       2.974        5.552       2.026
```

Note that the by argument is `iris[5]` (a list, or a data frame column) and not `iris[,5]` (a vector or factor)

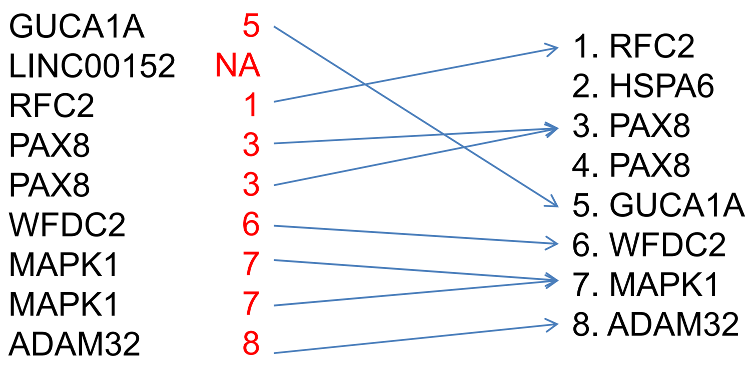
Merge: combining two data frames

```
> clindata
  patient age weight
1     i04  30     96
2     i06  35     98
3     i27  43     87
4     i32  57     85
5     i52  28     62
> genedata
  patient ESR1expr BRCA1expr
1     i04 7.411949  11.99540
2     i08 7.353114  12.43524
3     i27 8.374046  12.98381
4     i32 7.768207  11.76007
5     i52 8.539683  12.55489

> merge(clindata, genedata)
  patient age weight ESR1expr BRCA1expr
1     i04  30     96 7.411949  11.99540
2     i27  43     87 8.374046  12.98381
3     i32  57     85 7.768207  11.76007
4     i52  28     62 8.539683  12.55489
```

Merge: combining two data frames

```
> merge( clindata, genedata, all=TRUE )
  patient age weight ESR1expr BRCA1expr
1     i04  30     96 7.411949  11.99540
2     i06  35     98       NA        NA
3     i27  43     87 8.374046  12.98381
4     i32  57     85 7.768207  11.76007
5     i52  28     62 8.539683  12.55489
6     i08  NA     NA 7.353114  12.43524
```

Match: a general way for finding common values



```
> newlist <- c("GUCA1A", "LINC00152", "RFC2", "PAX8", "PAX8",
               "WFDC2", "MAPK1", "MAPK1", "ADAM32")
> reflist <- c("RFC2", "HSPA6", "PAX8", "PAX8", "GUCA1A",
               "WFDC2", "MAPK1", "ADAM32" )

> match(newlist, reflist)
[1]  5 NA  1  3  3  6  7  7  8
> newlist %in% reflist
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Character manipulations

Basic functions for character manipulation in R

```
> string <- paste("This", "is", "a", "string")
> string
[1] "This is a string"
> paste("This", "is", "a", "string", sep="-")
[1] "This-is-a-string"
> nchar(string)
16
> substring(string, 6, 7)
[1] "is"

> paste( "chr", c(1:22,"X", "Y"), sep="")
 [1]  "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"
 [8]  "chr8"  "chr9" "chr10" "chr11" "chr12" "chr13" "chr14"
[15] "chr15" "chr16" "chr17" "chr18" "chr19" "chr20" "chr21"
[22] "chr22"  "chrX"  "chrY"
> paste0( "chr", c(1:22,"X", "Y"))  # Same result
```

`strsplit()`*: splitting a string according to presence of a substring*

```
> transcript <-
     "NST00000293272(14),ENST00000366113(14),NM_002985(14)"

> strsplit(transcript, ",")
[[1]]
[1] "NST00000293272(14)"  "ENST00000366113(14)" "NM_002985(14)"
```

`strsplit()`: *splitting a string according to presence of a substring*

```
# From Affymetrix annotations:
> genesymbols <- "LOC441259 /// POLR2J2 /// POLR2J3 /// UPK3BL"

> strsplit(genesymbols, " /// ")
[[1]]
[1] "LOC441259" "POLR2J2"   "POLR2J3"   "UPK3BL"

> genesymbols <- "LOC441259"
> strsplit(genesymbols, " /// ")
[[1]]
[1] "LOC441259"
```

Splitting a string at all possible positions

```
> sequence <- "ATGCTCTCTGAAAACGTT"
> strsplit(sequence, "")              # We split on the empty string
[[1]]
 [1] "A" "T" "G" "C" "T" "C" "T" "C" "T" "G" "A" "A" "A" "A" "C" "G" "T" "T"

> strsplit(sequence, "")[[1]]
 [1] "A" "T" "G" "C" "T" "C" "T" "C" "T" "G" "A" "A" "A" "A" "C" "G" "T" "T"
> table( strsplit(sequence, "")[[1]] )

A C G T
5 4 3 6
```

R includes several functions for matching strings using regular expressions :

- `grep()` : find if a string contains a given pattern (see also `regexpr()`)
- `sub()` : find a pattern in a string and replace it (see also `gsub()`)

```
> genedata
  patient exprESR1 exprBRCA1
1     i04 7.411949  11.99540
2     i08 7.353114  12.43524
3     i27 8.374046  12.98381
4     i32 7.768207  11.76007
5     i52 8.539683  12.55489
> grep( "^expr", names(genedata) )
[1] 2  3
> genedata[ , grep( "^expr", names(genedata) ) ]
  exprESR1 exprBRCA1
1 7.411949  11.99540
2 7.353114  12.43524
3 8.374046  12.98381
4 7.768207  11.76007
5 8.539683  12.55489
```

Substitutions

```
> locations <- c("chr6p21.3", "chr7q11.23", "chr1q23",
                 "chr2q13",   "chr6p21.1", "chr3p21",
                 "chr17q11.2-q12", "chr10q24.3-qter")
> sub("^chr([0-9]+).+", "\\1", locations )
[1] "6" "7" "1" "2" "6" "3" "17" "10"

> locations <- c("chr6p21.3", "chr7q11.23", "chr1q23",
                 "chr2q13", "chr6p21.1", "chr3p21",
                 "chr17q11.2-q12", "chrXq26.3")
> sub("^chr([0-9]).+", "\\1", locations )
[1] "6"         "7"         "1"         "2"         "6"
[6] "3"        "17" "chrXq26.3"
```

Assign and retrieve variables «on the fly»

```
> assign("x" , mean(runif(10)) )
> get("x")
[1] 0.330505

> patientid <- "10"
> assign( paste("treatment", patientid, sep=""),
          sample( c("control", "treatment"), 1) )
> treatment10
[1] "treatment"
```

Use eval to create R commands «on the fly»

```
for (i in 1:12) {
  eval( parse(text=paste("temp.",i," <- c(",i,",",i,")",
                               sep="")))
}

> ls()
 [1] "i"       "temp.1"  "temp.10" "temp.11" "temp.12"
 [6] "temp.2"  "temp.3"  "temp.4"  "temp.5"  "temp.6“
[11] "temp.7"  "temp.8"  "temp.9"
> temp.1
[1] 1 1
> temp.10
[1] 10 10
```

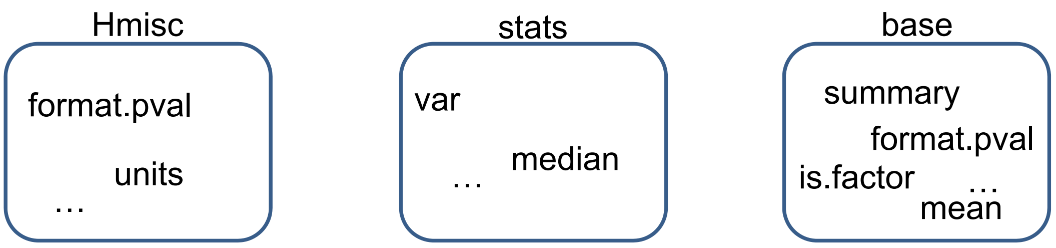
Namespaces

What happens when several packages define the same function?

Example: the Hmisc package

```
> library(Hmisc)
Loading required package: lattice
Loading required package: survival
Loading required package: Formula
Loading required package: ggplot2

Attaching package: ‘Hmisc’

The following objects are masked from ‘package:base’:

    format.pval, round.POSIXt, trunc.POSIXt, units
```

Each R package has its own namespace



When looking for a function, R follows a **search path** through the namespaces until it finds the first occurrence of the function it is looking for:

```
> search()
 [1] ".GlobalEnv"        "package:Hmisc"     "package:ggplot2"
 [4] "package:Formula"   "package:survival"  "package:lattice"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:utils"     "package:datasets"  "package:methods"
[13] "Autoloads"         "package:base"
```

Several packages can provide the same function, without any interference.

Functions from different packages can be differentiated using ::

```
> Hmisc::format.pval(0.05)
[1] "0.05"
> base::format.pval(0.05)
[1] "0.05"
```

This allows the redefinition of a function, still allowing access to its original version:

```
# My own summary
summary.default <- function( data ) {
  # Start by getting the original summary
  originalsummary <- base::summary.default(data)

  # Then we modify the output as we want
  …
}
```

Deleting the new function will let the original one available.

Example: redefining the addition

```
> 1+1
[1] 2
> `+` <- function(x, y) { base::`+`( base::`+`(x, y), 0.1 )  }
> 1+1
[1] 2.1
> rm(`+`)   # Don't forget to go back to a "sane" version of
the addition.
> 1+1
[1] 2
```

Exporting function

A package can choose to make a function available outside its namespace by exporting it.

Otherwise, by default, the code is only available to other functions from this package.

Example: the t.test function in package stats

```
> t.test
function (x, ...)
UseMethod("t.test")
<bytecode: 0x55ccd563e0c0>
<environment: namespace:stats>

> methods(t.test)
[1] t.test.default* t.test.formula*
see '?methods' for accessing help and source code
> t.test.default
Error: object 't.test.default' not found
```

The package exports t.test (which is then available from outside) but not t.test.default, which you are supposed to call through t.test only.

How to access a non-exported function ?

To get the source code:

```
> getAnywhere(t.test.default)
A single object matching ‘t.test.default’ was found
It was found in the following places
  registered S3 method for t.test from namespace stats
  namespace:stats
with value

function (x, y = NULL, alternative = c("two.sided", "less",
"greater"),
    mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95,
    ...)
{
    alternative <- match.arg(alternative)
    if (!missing(mu) && (length(mu) != 1 || is.na(mu)))
```

How to access a non-exported function ?

To run it:

```
> stats::t.test.default()
Error: 't.test.default' is not an exported object from
'namespace:stats'

> stats:::t.test.default()
Error in stats:::t.test.default() :
  argument "x" is missing, with no default
```

However, if a function is not exported, there is usually a good reason.

R best practices

What we are not going to discuss: style rules

- How to format your code
- How to indent your code
- How to name your identifiers
- How to use comments
- …

Suggestion of styleguides:

- Google styleguide: https://google.github.io/styleguide/Rguide.xml
- Hadley Wickham's R style guide: http://adv-r.had.co.nz/Style.html

<-

vs

=

Assignment operators: <- *vs* =

- R provides 5 assignment operators:

```
?assignOps
    Description
    Assign a value to a name.
    Usage
    x <- value
    x <<- value
    value -> x
    value ->> x
    x = value
```

- We will discuss <<- later

- -> and ->> allow the assignment to be done left to right (something impossible with =)

Assignment operators: <- *vs* =

- Originally, R would only accept <- for assignment
- This choice has a historical origin in the APL programming language, at a time where "←" was an actual key on the keyboard
- The "=" operator was added in 2001, for improving compatibility with other languages.

- Both Hadley Wickam's and Google's styleguides suggest using "<-" only, and so does the R community in general

- The two operators are mostly interchangeable
- There are a few exceptions, though…

Assignment operators: <- *vs* =

- Function parameters can only be specified with an "=":

```
    mean(data, na.rm=TRUE)    # works
    mean(data, na.rm<-TRUE)   # does not work
```

- However, if you want to specify an assignment within a parameter, you must use <-
- For example, if you want to compute an expression, store it and measure its execution time simultaneously:

```
    system.time(result<-expression) # works
```

Assignment operators: <- *vs* =

- Using `result=expression` would not work, as the `system.time()` function does not accept a `result` parameter
- An alternative way of doing this would be:

```
    system.time( (result=expression) )
```

- More generally, <- can be used everywhere, while = can only be used at the "top level"
- For example:

```
    if (x <- 0) 1 else 0     # works
    if (x = 0)  1 else 0     # does not work
```

- One reason for this: confusing `x=0` and `x==0` is one of the most common mistake in other programming languages
- But in most cases, you can probably avoid using such a construct anyway…

Local vs global variables

```
> m <- 1
> f <- function() { m <- m + 1 }
> f()
> m
[1] 1
```

Local vs global variables

```
> m <- 1
> f <- function() { m <<- m + 1 }
> f()
> m
[1] 2
```

The "<<-" operator forces the assignment to work on the global m variable, and not on a local variable that exists only inside the loop.

```
> sample(1:100, 10, replace=T)
[1] 27 38 58 91 21 90 95 67 63
```

```
> A <- "a"; B <- "b"; C <- "c"; T <- "t"


> sample(1:100, 10, replace=T)
> A <- "a"; B <- "b"; C <- "c"; T <- "t"


> sample(1:100, 10, replace=T)
Error in sample(1:100, 10, replace = T) : invalid 'replace' argument
```

‘T’ and ‘F’, as shortcuts for TRUE and FALSE, can freely be redefined by the user, something impossible with the full form:

```
> TRUE <- "t"
Error in TRUE <- "t" : invalid (do_set) left-hand side to assignment
```

This will yield an error, or even worse…

If you are really vicious…

```
> T <- FALSE

> sample(1:10, 10, replace=T)
 [1]  7  6  3  4 10  1  8  5  9  2


# What will happen, more likely:
> T <- complicated_function( many, many, complicated, arguments, and
                             the, function, returns, FALSE, in, the,
                             end )
> sample( 1:10, 10, replace=T )
 [1]  7  6  3  4 10  1  8  5  9  2
```

```
selectcolumns <- function( m, cols, rows ) {
  m1 <- m [, cols]
  m2 <- m1[rows, ]
  m2
}

nrows <- 20
m1 <- data.frame( a=runif(nrows), b=runif(nrows), c=runif(nrows) )
row.names(m1) <- paste( "row", 1:nrow(m1), sep="")

cols <- c("b", "c")
rows <- c("row10", "row12")

> selectcolumns(m1, cols, rows)
              b         c
row10 0.8578518 0.2864960
row12 0.3767570 0.7874534
```

```
selectcolumns <- function( m, cols, rows ) {
  m1 <- m [, cols]
  m2 <- m1[rows, ]
  m2
}

nrows <- 20
m1 <- data.frame( a=runif(nrows), b=runif(nrows), c=runif(nrows) )
row.names(m1) <- paste( "row", 1:nrow(m1), sep="")

cols <- "b"
rows <- c("row10", "row12")

> selectcolumns(m1, cols, rows)
Error in m1[rows, ] : incorrect number of dimensions
```

By default, R removes all dimensions that it deems not useful:

```
> m <- matrix(1:4, nrow=2)
> m[,1:2]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

yields a matrix, but

```
> m[,1]
[1] 1 2
```

yields a vector (instead of 2 x 1 matrix).

To avoid this, use the `drop=FALSE` option to the matrix subsetting:

```
> m[,1]
[1] 1 2

> m[,1, drop=FALSE]
     [,1]
[1,]    1
[2,]    2

 > m[1,, drop=FALSE]
```

It is not possible to set `drop=FALSE` as the default mode.

Doing this would mean that accessing one element in a matrix would return a 1x1 matrix:

```
> M[2,3, drop=FALSE]
           [,1]
    [1,]      4
```

which is almost certainly not what you want.

Another possible consequence

```
> head(data1, 3)
  identifier    var1    var2
1       3862 0.87207 -2.0105
2       1577 0.01075  0.1970
3       5150 1.28249 -0.4650
> head(data2, 3)
  identifier   var3    var4
1       3862 0.1383 -2.0165
2       1577 2.3219  0.6855
3       5150 0.6865  0.7783
> data <- cbind( data1[, c("var1", "var2")],
                data2[, c("var3", "var4")], data1[, "identifier"] )
```

Matrices converted to vectors lose their names !

```
> head(data1, 3)
  identifier    var1    var2
1       3862 0.87207 -2.0105
2       1577 0.01075  0.1970
3       5150 1.28249 -0.4650
> head(data2, 3)
  identifier   var3    var4
1       3862 0.1383 -2.0165
2       1577 2.3219  0.6855
3       5150 0.6865  0.7783
> data <- cbind( data1[, c("var1", "var2")],
                 data2[, c("var3", "var4")], data1[, "identifier"] )
> head(data, 3)
       var1     var2     var3    var4 data1[, "identifier"]
1   0.87207 -2.01057  0.13836 -2.0165                  3862
2   0.01075  0.19709  2.32192  0.6855                  1577
3   1.28249 -0.46507  0.68659  0.7783                  5150
```

Avoid the `attach` *command*

```
# Starting from a clean R session
> data <- list( a=1, b=2 )
> attach(data)
> a
[1]   1
# equivalent to
> data$a
[1]   1
```

Avoid the `attach` *command*

```
> a <- 0; data <- list(a=1, b=2)       # a = 0
> attach(data)                         # a = ?
# Warning displayed
> a <- 3                               # a = ?  data$a = ?
> rm(a)                                # a = ?
> data$a <- 4                          # a = ?
> attach(data)                         # a = ?
# Warning message displayed
> rm(a)                                # a = ?
> detach(data)                         # a = ?
> detach(data)                         # a = ?
> attach(data)                         # a = ?
> rm(list = ls())                      # a = ?
> detach(data)                         # a = ?
```

Avoid the `attach` *command*

```
> a <- 0; data <- list(a=1, b=2)       # a = 0
> attach(data)                         # a = 0
# Warning displayed
> a <- 3                               # a = 3  data$a = 1
> rm(a)                                # a = 1
> data$a <- 4                          # a = 1
> attach(data)                         # a = 4
# Warning message displayed
> rm(a)                                # a = 4  (error message)
> detach(data)                         # a = 1
> detach(data)                         # Error message
> attach(data)                         # a = 4
> rm(list = ls())                      # a = 4
> detach(data)                         # Error message
```

*Use «***`with`***», «*`within`*» or «*`transform`*» instead*

```
> head(clinicaldata, 3)
  phenotype  genotype
1 0.8142518 0.9347601
2 0.9287772 0.3461621
3 0.1474810 0.5330606

> with( clinicaldata,  plot( genotype, phenotype ) )

# Equivalent to
> plot( clinicaldata$genotype, clinicaldata$phenotype )
```

Use «`with`*», «***`within`***» or «*`transform`*» instead*

```
> head(clinicaldata, 3)
  phenotype  genotype
1 0.8142518 0.9347601
2 0.9287772 0.3461621
3 0.1474810 0.5330606

> new <- within(clinicaldata,  genotype <- log2(genotype)))
> new
 phenotype    genotype
1 0.8142518 -0.09733194
2 0.9287772 -1.53048032
3 0.1474810 -0.90762854
```

*Use «`with`», «`within`» or «**`transform`**» instead*

```
> head(clinicaldata, 3)
  phenotype  genotype
1 0.8142518 0.9347601
2 0.9287772 0.3461621
3 0.1474810 0.5330606

> transform(clinicaldata, genotype = log2( genotype))

# Equivalent to
> clinicaldata$genotype <- log2(clinicaldata$genotype)
```

Using `transform()` is clearer than using the direct command, but less flexible than using `within()`.

Subset

```
> head(clinicaldata, n=3)
  phenotype age sex weight
1  4.373546  NA   F     77
2  5.183643  46   M     89
3  4.164371  52   M     76

> subset(clinicaldata, sex=="F" & age <40, select=-weight)
   phenotype age sex
8   5.738325  39   F
16  4.955066  24   F
17  4.983810  32   F
20  5.593901  36   F
```

The subset commands allows the selection of rows (or elements of vectors) based on logical expressions, and selection of columns based on names.

It removes NA values from columns where a selection is done.

Subset

The `subset` function is useful when working in an interactive session, but its use is not recommended in scripts, according to the help page:

```
Warning:

  This is a convenience function intended for
  use interactively. For programming it is
  better to use the standard subsetting
  functions like ‘[’, and in particular the
  non-standard evaluation of argument ‘subset’
  can have unanticipated consequences.
```

✓ Reproducible Research

How can we improve this code ?

```
> annotations <- read.table("annotations-from-provider.txt")
   identifier entrezid gene
1          31    73398    H
2          41    55359    P
3          89    97377    H
4          63    37348    Y
5          17     4465    T
6          55    55583    Z
7          55    17866    K
...
# We do not need the gene code
> annotations <- annotations[,1:2]
```

If available, always use data frame names instead of column numbers:

```
> annotations <- annotations[, c("identifier", "entrezid") ]
```

Loading file affy-annot.txt into R

```
   ProbeSet ID    ID        Target Description
1  1007_s_at      U48705    discoidin domain receptor tyrosine kinase 1
2  1053_at        M87338    replication factor C (activator 1) 2, 40kDa
3  117_at         X51757    heat shock 70kDa protein 6 (HSP70B')
4  121_at         X69699    paired box 8
5  1255_g_at      L36861    guanylate cyclase activator 1A (retina)
6  1294_at        L13852    ubiquitin-like modifier activating enzyme 7
7  1487_at        L38487    Human ER-related protein (hERRa1) mRNA, 3' end
8  1316_at        X55005    thyroid hormone receptor, alpha
9  1320_at        X79510    protein tyrosine phosphatase, non-receptor type 21
10 1405_i_at      M21121    chemokine (C-C motif) ligand 5
11 1431_at        J02843    cytochrome P450, family 2, subfamily E, polypeptide 1
12 1438_at        X75208    EPH receptor B3
```

```
> data <- read.table("affy-annot.txt", sep="\t")
> dim(data)
[1] 8 3
```

Where are the 4 missing rows?

Loading file affy-annot.txt into R

```
   ProbeSet ID    ID        Target Description
1  1007_s_at      U48705    discoidin domain receptor tyrosine kinase 1
2  1053_at        M87338    replication factor C (activator 1) 2, 40kDa
3  117_at         X51757    heat shock 70kDa protein 6 (HSP70B')
4  121_at         X69699    paired box 8
5  1255_g_at      L36861    guanylate cyclase activator 1A (retina)
6  1294_at        L13852    ubiquitin-like modifier activating enzyme 7
7  1487_at        L38487    Human ER-related protein (hERRa1) mRNA, 3' end
8  1316_at        X55005    thyroid hormone receptor, alpha
9  1320_at        X79510    protein tyrosine phosphatase, non-receptor type 21
10 1405_i_at      M21121    chemokine (C-C motif) ligand 5
11 1431_at        J02843    cytochrome P450, family 2, subfamily E, polypeptide 1
12 1438_at        X75208    EPH receptor B3
```

```
> data <- read.table("affy-annot.txt", sep="\t")
> dim(data)
[1] 8 3
```

Where are the 4 missing rows?

Loading file affy-annot.txt into R

```
   ProbeSet ID    ID       Target Description
1  1007_s_at      U48705   discoidin domain receptor tyrosine kinase 1
2  1053_at        M87338   replication factor C (activator 1) 2, 40kDa
3  117_at         X51757   heat shock 70kDa protein 6 (HSP70B'...' end
8  1316_at        X55005   thyroid hormone receptor, alpha
9  1320_at        X79510   protein tyrosine phosphatase, non-receptor type 21
10 1405_i_at      M21121   chemokine (C-C motif) ligand 5
11 1431_at        J02843   cytochrome P450, family 2, subfamily E, polypeptide 1
12 1438_at        X75208   EPH receptor B3
```

```
> data <- read.table("affy-annot.txt", sep="\t")
> dim(data)
[1] 8 3
```

The 4 missing lines are all in the 3rd row.

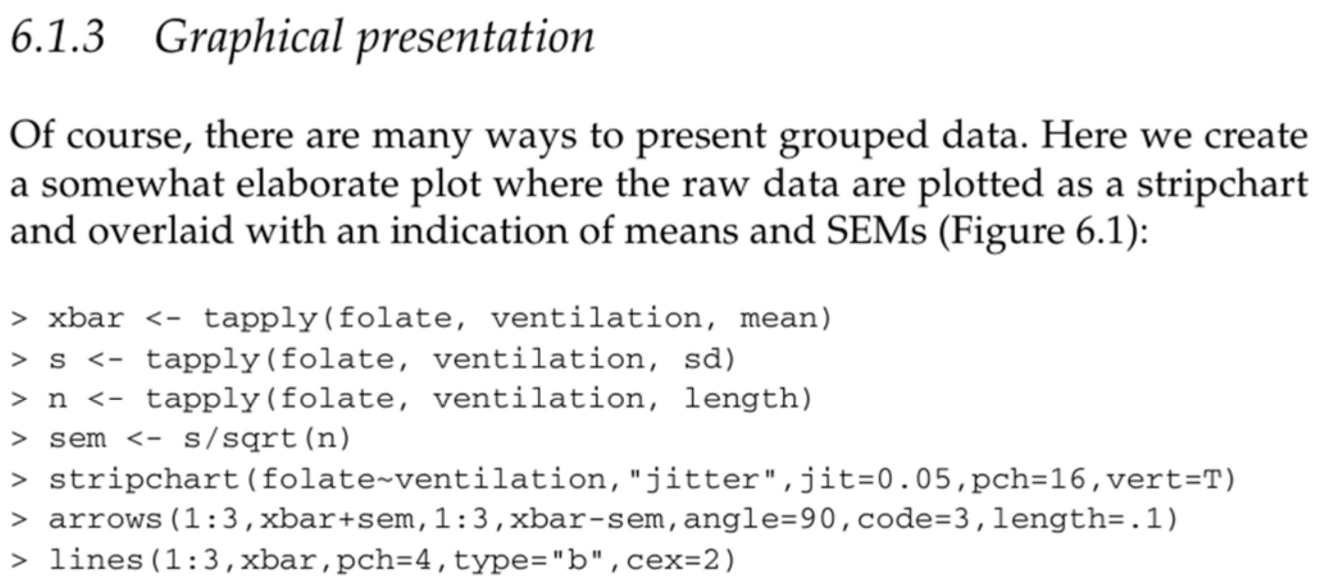
✓ Reproducible Research

- Assume as little as possible about your data

- In particular, always specify the «`quote`» argument when reading a file (especially if you do not use quotes !):

```
> data <- read.table("affy-annot.txt", sep="\t", quote="" )
> dim(data)
[1] 12  3
```

Reproducible
Research



6.1.3 *Graphical presentation*

Of course, there are many ways to present grouped data. Here we create a somewhat elaborate plot where the raw data are plotted as a stripchart and overlaid with an indication of means and SEMs (Figure 6.1):

```
> xbar <- tapply(folate, ventilation, mean)
> s <- tapply(folate, ventilation, sd)
> n <- tapply(folate, ventilation, length)
> sem <- s/sqrt(n)
> stripchart(folate~ventilation,"jitter",jit=0.05,pch=16,vert=T)
> arrows(1:3,xbar+sem,1:3,xbar-sem,angle=90,code=3,length=.1)
> lines(1:3,xbar,pch=4,type="b",cex=2)
```

P. Dalgaard, «Introductory Statistics with R» (1st edition, 2002), p. 118

The code does not work in R 2.15

```
> library(ISwR)
> data(red.cell.folate)
> attach(red.cell.folate)
> stripchart(folate~ventilation, "jitter", jit=0.05, pch=16,
+ vert=T)
Error in eval(predvars, data, env) : invalid 'envir' argument
```

```
               CHANGES IN R VERSION 2.7.0


GRAPHICS CHANGES


   o   stripchart() is now a generic function, with default and
       formula methods defined.  Additional graphics parameters may
       be included in the call.  Formula handling is now
       similar to boxplot().
```

```
stripchart              package:graphics              R Documentation

1-D Scatter Plots

Description:

     'stripchart' produces one dimensional scatter plots (or dot plots)
     of the given data.  These plots are a good alternative to
     'boxplot's when sample sizes are small.

Usage:

     stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
                vertical = FALSE, group.names, add = FALSE,
                at = NULL, xlim = NULL, ylim = NULL,
                ylab=NULL, xlab=NULL, dlab="",
                log = "", pch = 0, col = par("fg"), cex = par("cex")

Arguments:

       x: the data from which the plots are to be produced.  The data
          can be specified as a single numeric vector, or as list of
          numeric vectors, each corresponding to a component plot.
          Alternatively a symbolic specification of the form ´x ~ g´
          can be given, indicating the observations in the vector ´x´
          are to be grouped according to the levels of the factor ´g´.
```

R 2.6

```
stripchart(folate~ventilation, "jitter", jit=0.05,pch=16,vert=T)
```

✓ ✓

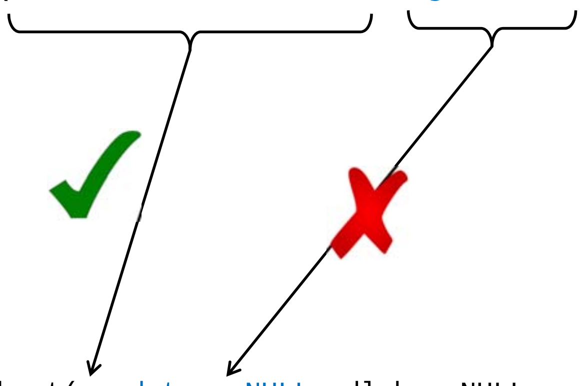
```
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
           vertical = FALSE, group.names, add = FALSE,
           at = NULL, xlim = NULL, ylim = NULL,
           ylab=NULL, xlab=NULL, dlab="",
           log = "", pch = 0, col = par("fg"), cex = par("cex“)

Arguments:

       x: the data from which the plots are to be produced.  The data
          can be specified as a single numeric vector, or as list of
          numeric vectors, each corresponding to a component plot.
          Alternatively a symbolic specification of the form ´x ~ g´
          can be given, indicating the observations in the vector ´x´
          are to be grouped according to the levels of the factor ´g´.
```

R 2.6

```
stripchart              package:graphics              R Documentation

1-D Scatter Plots

Description:

     ‘stripchart’ produces one dimensional scatter plots (or dot plots)
     of the given data.  These plots are a good alternative to
     ‘boxplot’s when sample sizes are small.

Usage:

     stripchart(x, ...)

     ## S3 method for class 'formula'
     stripchart(x, data = NULL, dlab = NULL, ...,
                subset, na.action = NULL)


     ## Default S3 method:
     stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
                vertical = FALSE, group.names, add = FALSE,
                at = NULL, xlim = NULL, ylim = NULL,
                ylab=NULL, xlab=NULL, dlab="", glab="",
                log = "", pch = 0, col = par("fg"), cex = par("cex"),
                axes = TRUE, frame.plot = axes, ...)
```

R 2.7

```
stripchart(folate~ventilation, "jitter", jit=0.05,pch=16,vert=T)
```



```
stripchart(x, data = NULL, dlab = NULL, ...,
              subset, na.action = NULL)


## Default S3 method:
stripchart(x, method = "overplot", jitter = 0.1, offset = 1/3,
              vertical = FALSE, group.names, add = FALSE,
              at = NULL, xlim = NULL, ylim = NULL,
              ylab=NULL, xlab=NULL, dlab="", glab="",
              log = "", pch = 0, col = par("fg"), cex = par("cex"),
              axes = TRUE, frame.plot = axes, ...)
```

R 2.7

Corrected code in the second edition of the book

```
> library(ISwR)
> data(red.cell.folate)
> attach(red.cell.folate)
> stripchart(folate~ventilation, "jitter", jit=0.05, pch=16,
+ vert=T)
Error in eval(predvars, data, env) : invalid 'envir' argument

> stripchart(folate~ventilation, method="jitter",
+ jitter=0.05, pch=16, vert=T)
```

Also worth noting: the short parameter «`jit`» has been replaced by the full name «`jitter`»

P. Dalgaard, «Introductory Statistics with R» (2nd edition, 2008), p. 134

Reproducible
Research

```
             CHANGES IN R VERSION 2.4.0


USER-VISIBLE CHANGES

    o   The functions read.csv(), read.csv2(), read.delim(),
        read.delim2() now default their 'comment.char' argument to "".
        (These functions are designed to read files produced by other
        software, which might use the # character inside fields, but
        are unlikely to use it for comments.)
```

Reproducible
Research

Storing the session information

```
> library(affy)
> sessionInfo()
R version 2.15.1 (2012-06-22)
Platform: x86_64-pc-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_AU.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_AU.UTF-8        LC_COLLATE=en_AU.UTF-8
 [5] LC_MONETARY=en_AU.UTF-8    LC_MESSAGES=en_AU.UTF-8
 [7] LC_PAPER=C                 LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_AU.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] affy_1.34.0        Biobase_2.16.0     BiocGenerics_0.2.0

loaded via a namespace (and not attached):
[1] affyio_1.24.0         BiocInstaller_1.4.7   preprocessCore_1.18.0
[4] zlibbioc_1.2.0
```

Reproducible
Research

Storing the session information

```
> sink("logfile.txt")
> sessionInfo()
...
> sink()

> capture.output( sessionInfo(), file="logfile.txt" )
> log <- capture.output( sessionInfo() )
 [1] "R version 2.15.1 (2012-06-22)"
 [2] "Platform: x86_64-pc-linux-gnu (64-bit)"
 [3] ""
 [4] "locale:"
 [5] " [1] LC_CTYPE=en_AU.UTF-8       LC_NUMERIC=C              "
 [6] " [3] LC_TIME=en_AU.UTF-8        LC_COLLATE=en_AU.UTF-8    "
 [7] " [5] LC_MONETARY=en_AU.UTF-8    LC_MESSAGES=en_AU.UTF-8   "
...
```

Use sink() and capture.output() .

What could go wrong in this code ?

```
n <- 100
results <- rep(0, n)
for (i in 1:n) {
  data <- read.table(paste("data", i, ".txt", sep=""))
  model <- lm( data$y ~ data$x )
  results[i] <- coef(model)[2,1]
}
```

Managing errors

```
> n <- 100
> results <- rep(NA, n)
> for (i in 1:n) {
+   data <- read.table(paste("data", n, ".txt", sep=""))
+   model <- lm( data$y ~ data$x )
+   results[i] <- coef(model)[2,1]
+ }
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'data100.txt': No such file or directory
```

A potential problem if you don't check for errors

```
data <- read.table("data1")
# Do something with the data
...
data <- read.table("data2")
# Do something with the data
...
```

If you execute this code interactively (e.g. by pasting it in an R console) and the second `read.table()` call fails and you miss the error, then the `data` variable will still contain the content of file "data1", so that the rest of the code will seem to work ok.

Error handling: `try()`

```
n <- 100
results <- rep(0, n)
for (i in 1:n) {
  data <- try( read.table(paste("data", i, ".txt", sep="")) )
  if ( inherits(data, "try-error")) {
     results[i] <- NA
  } else {
    model <- lm( data$y ~ data$x )
    results[i] <- coef(model)[2,1]
  }
}
```

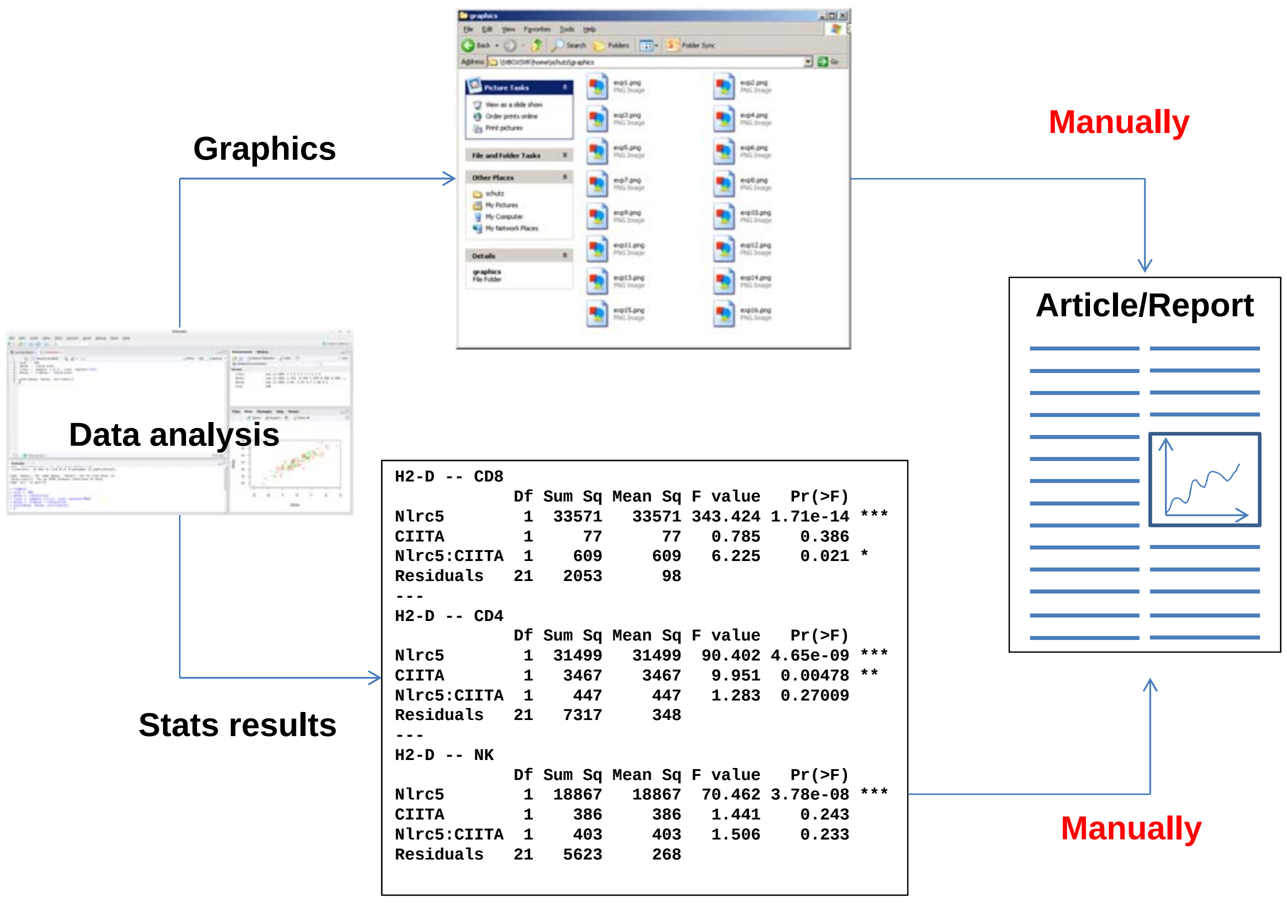
See `try()` and `tryCatch()`

```
# Generate a dataset
set.seed(1)
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
data <- data.frame(x, y)

# Fit a linear model
model <- lm( data$y ~ data$x )

# Generate a second dataset
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
newdata <- data.frame(x, y)

# Use the linear model to perform a prediction on the newdata
predict(model, newdata)
```

This code does not return any error message, but it does not work. Why ?

```
# Generate a dataset
set.seed(1)
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
data <- data.frame(x, y)

# Fit a linear model
model <- lm( data$y ~ data$x )

# Generate a second dataset
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
newdata <- data.frame(x, y)

# Use the linear model to perform a prediction on the newdata
predict(model, newdata)
```

This code does not return any error message, but it does not work. Why ?



```
H2-D -- CD8
            Df Sum Sq Mean Sq F value   Pr(>F)
Nlrc5        1  33571   33571 343.424 1.71e-14 ***
CIITA        1     77      77   0.785    0.386
Nlrc5:CIITA  1    609     609   6.225    0.021 *
Residuals   21   2053      98
---
H2-D -- CD4
            Df Sum Sq Mean Sq F value   Pr(>F)
Nlrc5        1  31499   31499  90.402 4.65e-09 ***
CIITA        1   3467    3467   9.951  0.00478 **
Nlrc5:CIITA  1    447     447   1.283  0.27009
Residuals   21   7317     348
---
H2-D -- NK
            Df Sum Sq Mean Sq F value   Pr(>F)
Nlrc5        1  18867   18867  70.462 3.78e-08 ***
CIITA        1    386     386   1.441    0.243
Nlrc5:CIITA  1    403     403   1.506    0.233
Residuals   21   5623     268
```

Use knitr in order to prepare reports

Allows you to integrate your results in a report.

Write the R code directly with the text, andlater integrate the results directly into the text.

Knitr: http://yihui.name/knitr/

Dynamic documents with knitR

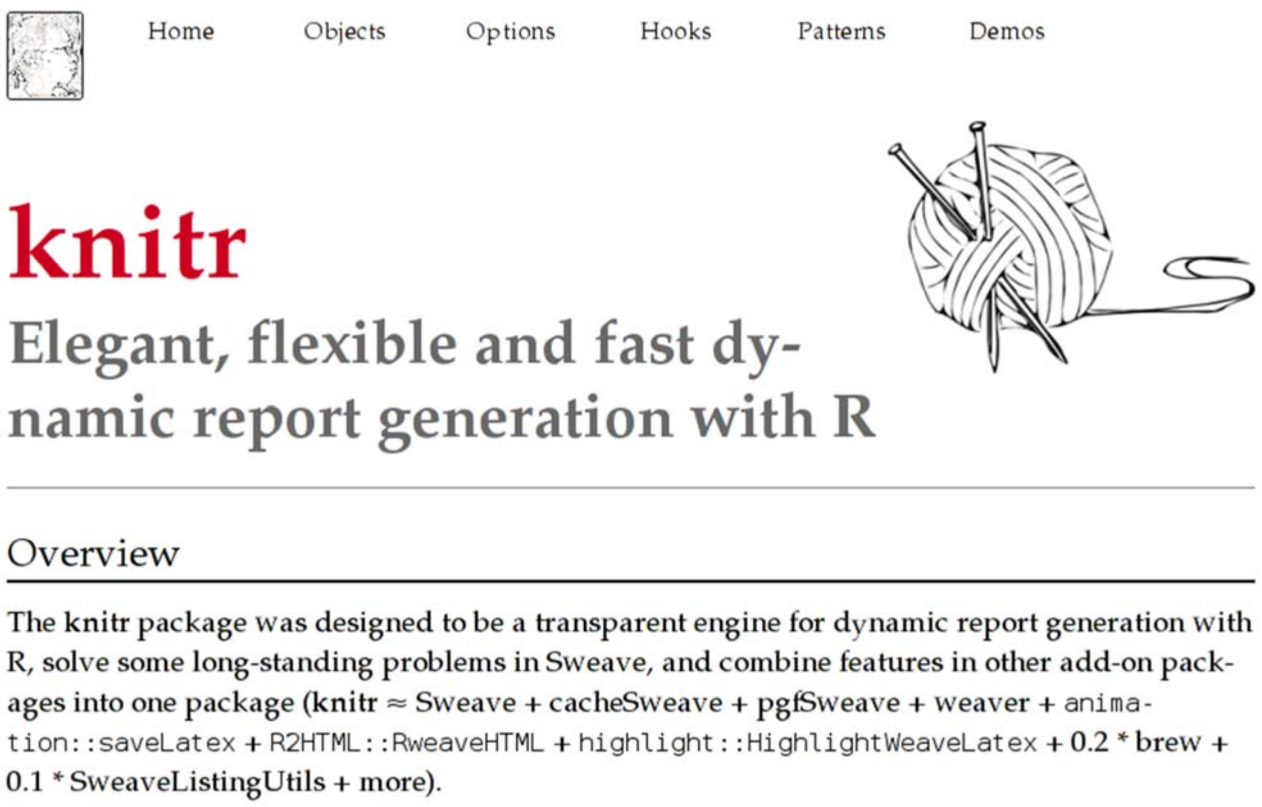
Dynamic documents

- Based on the idea of **literate programming**
- Combine program code and explanation/ documentation in same document (Donald Knuth, 1984)
- Documents in which the information is always up-to-date
- Write your report step by step while processing the data, in the same file

- Integrate your results in a report: *write the R code directly with the text, and later integrate the results directly into the text.*

Dynamic documents: software of interest

- Sweave: `http://www.stat.uni-muenchen.de/~leisch/Sweave/`
- **knitr** : `http://yihui.name/knitr/`
- LaTeX: `http://www.latex-project.org/`
- markdown: `http://daringfireball.net/projects/markdown/`



knitr

Elegant, flexible and fast dynamic report generation with R

Overview

The knitr package was designed to be a transparent engine for dynamic report generation with R, solve some long-standing problems in Sweave, and combine features in other add-on packages into one package (knitr ≈ Sweave + cacheSweave + pgfSweave + weaver + `animation::saveLatex` + `R2HTML::RweaveHTML` + `highlight::HighlightWeaveLatex` + 0.2 * brew + 0.1 * SweaveListingUtils + more).

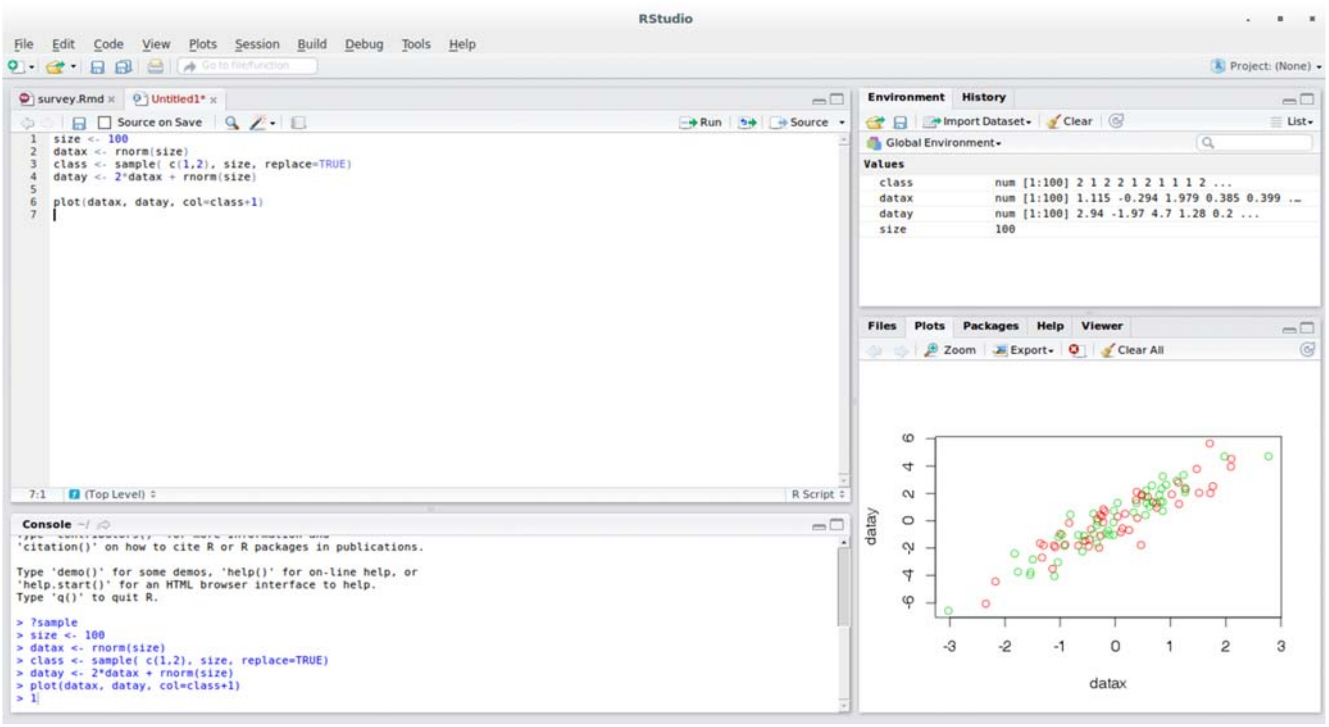
http://yihui.name/knitr/

Why use knitr ?

- all-in-one: analysis, documenting, formatting, reporting
- no annoying and error-prone copy-pasting
- modifying input data or code: changes are directly reflected in report
- easy to display underlying code in report when needed
- split code in chunks, but can still access all previously defined
- variables (single R session)
- flexible: code externalization, child documents, caching,...

- R
- `knitr` R package
- Editor with some support for R and configured to provide support for **knitr**
  RStudio is strongly suggested,
  otherwise see http://yihui.name/knitr/demo/editors/
- TeX Live (required for PDF output)
- pandoc
- learn from demos and examples:
  - `http://yihui.name/knitr/`
  - `http://rpubs.com`

RStudio



```
size <- 100
datax <- rnorm(size)
class <- sample( c(1,2), size, replace=TRUE)
datay <- 2*datax + rnorm(size)

plot(datax, datay, col=class+1)
```

```
Values
  class          num [1:100] 2 1 2 2 1 2 1 1 1 2 ...
  datax          num [1:100] 1.115 -0.294 1.979 0.385 0.399 ...
  datay          num [1:100] 2.94 -1.97 4.7 1.28 0.2 ...
  size           100
```

```
'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.

> ?sample
> size <- 100
> datax <- rnorm(size)
> class <- sample( c(1,2), size, replace=TRUE)
> datay <- 2*datax + rnorm(size)
> plot(datax, datay, col=class+1)
> 1
```

http://www.rstudio.com/products/rstudio/download/

How can we write the report ?

- Write .Rnw files, and generate PDF reports using LaTeX
- keep general structure of standard LATEX document:

```
\documentclass{...}
\usepackage{...}
\begin{document}
...
\end{document}
```

- Use the same LATEX packages/configurations as usual
- Add R chunks in the LaTeX code

If LaTeX is too scary, consider:

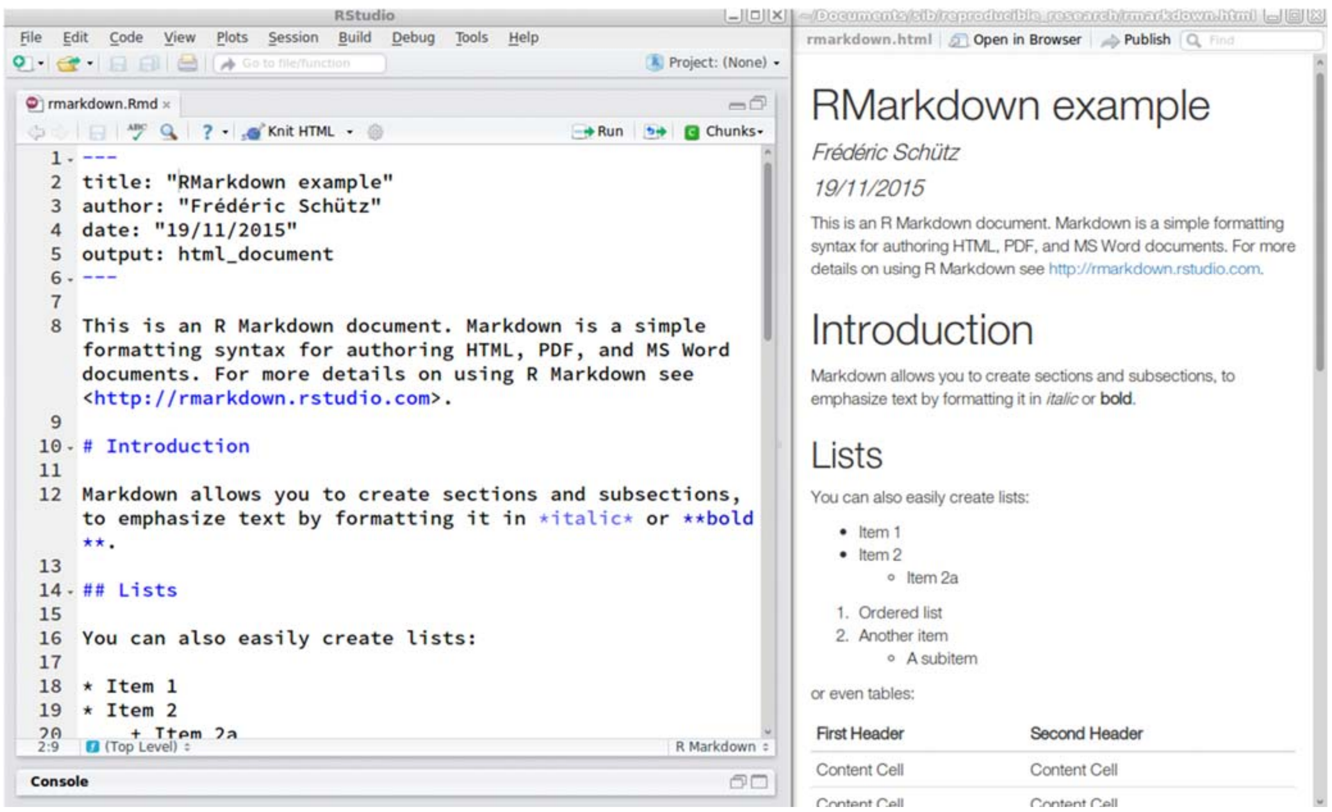
- LYX: `http://www.lyx.org/`
- markdown:
  - `http://www.rstudio.com/ide/docs/authoring/using_markdown`
  - `https://github.com/adam-p/markdown-here/wiki/`

Markdown and RMarkdown

Markdown is a simple plain text format that allows you to specify the layout of a document, and which can easily be converted to different formats afterwards.

R Markdown combines the core syntax of markdown (easy-to-write plain text format) with embedded R code chunks that are run so their output can be included in the final document.

R Markdown v2 (http://rmarkdown.rstudio.com/)



```
---
title: "RMarkdown example"
author: "Frédéric Schütz"
date: "19/11/2015"
output: html_document
---

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

# Introduction

Markdown allows you to create sections and subsections,
to emphasize text by formatting it in *italic* or **bold
**.

## Lists

You can also easily create lists:

* Item 1
* Item 2
    + Item 2a
```

RMarkdown example

Frédéric Schütz

19/11/2015

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

Introduction

Markdown allows you to create sections and subsections, to emphasize text by formatting it in *italic* or **bold**.

Lists

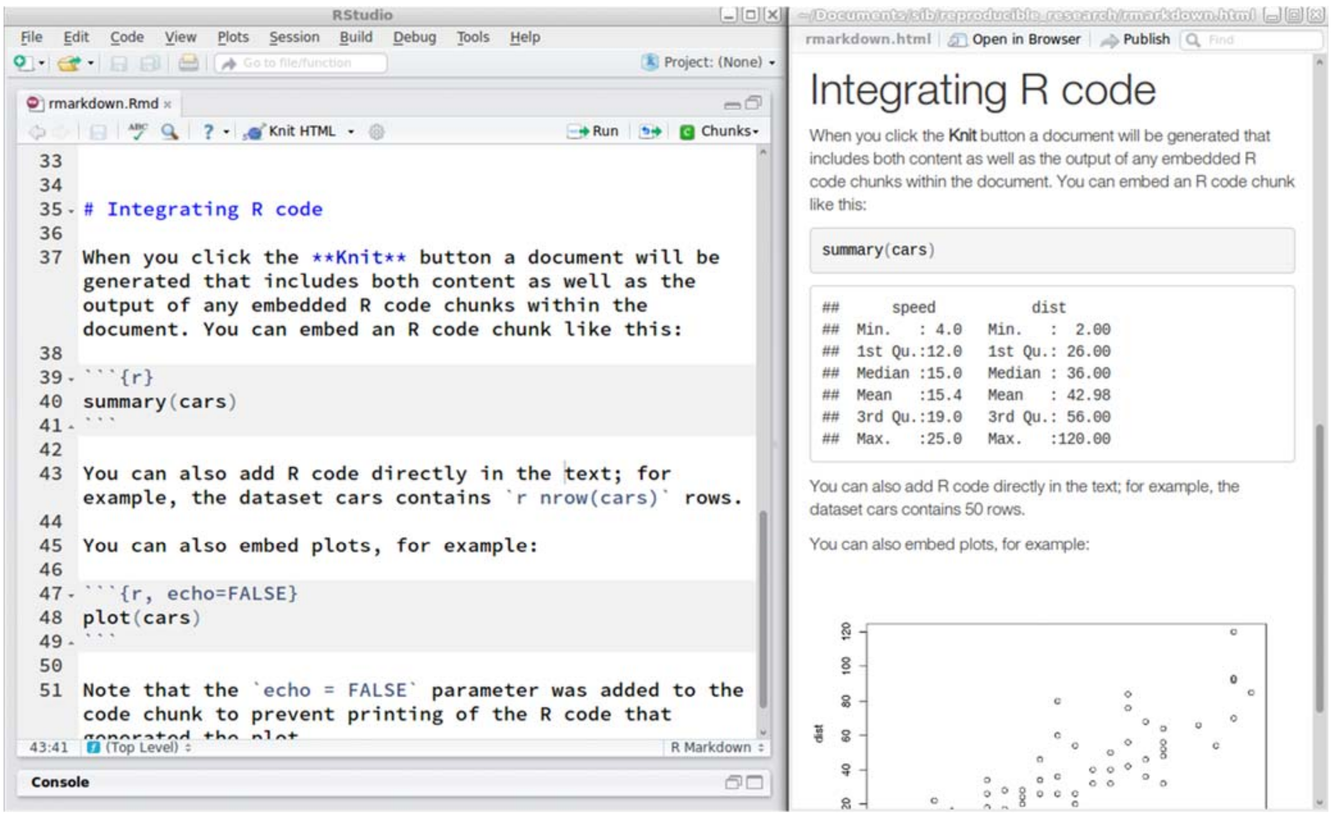
You can also easily create lists:

- Item 1
- Item 2
  - Item 2a

1. Ordered list
2. Another item
   - A subitem

or even tables:

First Header	Second Header
Content Cell	Content Cell
Content Cell	Content Cell



~~~
# Integrating R code

When you click the **Knit** button a document will be
generated that includes both content as well as the
output of any embedded R code chunks within the
document. You can embed an R code chunk like this:

```{r}
summary(cars)
~~~

You can also add R code directly in the text; for
example, the dataset cars contains `r nrow(cars)` rows.

You can also embed plots, for example:

```{r, echo=FALSE}
plot(cars)
```

Note that the `echo = FALSE` parameter was added to the
code chunk to prevent printing of the R code that
generated the plot
```

Integrating R code

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

You can also add R code directly in the text; for example, the dataset cars contains 50 rows.

You can also embed plots, for example:

dist
20
40
60
80
100
120

R Markdown example

```
---
title: "Untitled"
author: "Frédéric Schütz"
date: "23/01/2015"
output: html_document
---

This is an R Markdown document. Markdown is a simple
formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be
generated that includes both content as well as the output of
any embedded R code chunks within the document. You can embed
an R code chunk like this:
```
```

## *R Markdown example (continued)*

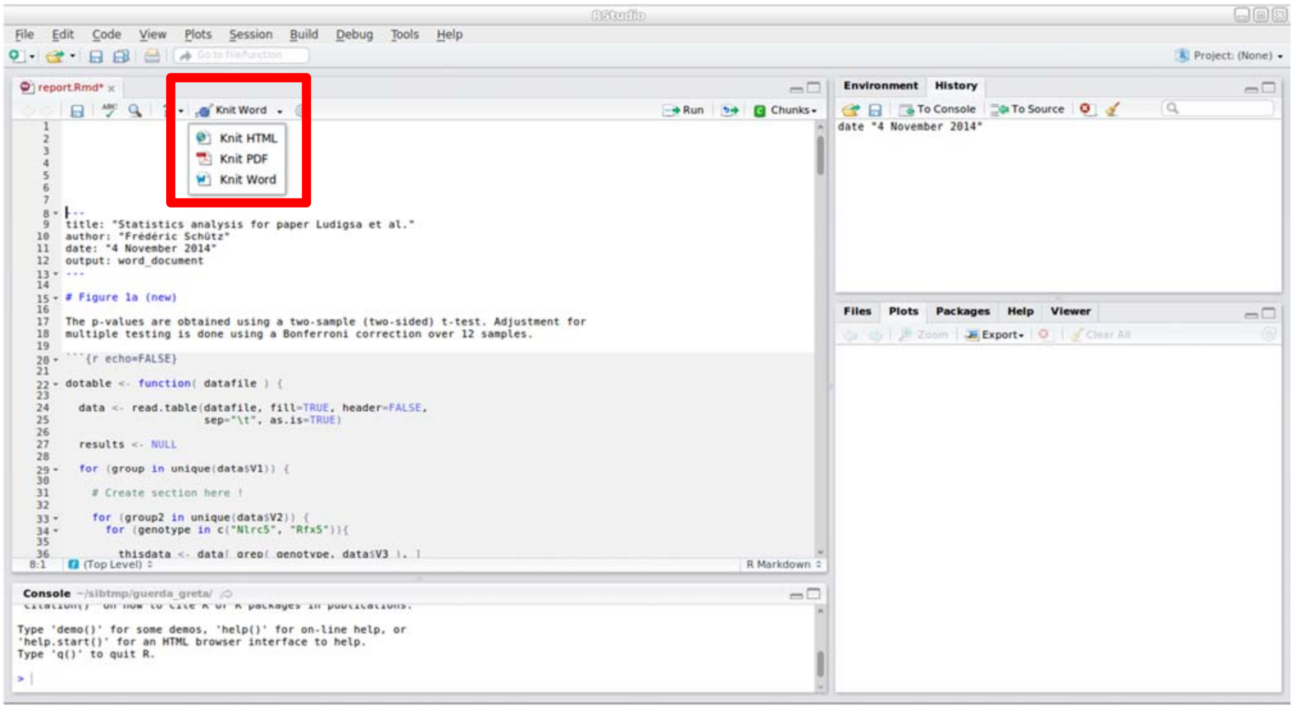
~~~
```{r}
summary(cars)
~~~

You can also embed plots, for example:

```{r, echo=FALSE}
plot(cars)
```

Note that the `echo = FALSE` parameter was added to the code
chunk to prevent printing of the R code that generated the
plot.
```

- Emphasis:

```
*italic*   **bold**
_italic_   __bold__
```

- Headers

```
# Header 1
## Header 2
### Header 3
```

- Unordered List:

```
* Item 1
* Item 2
    + Item 2a
    + Item 2b
```

- Ordered list:

```
1. Item 1
2. Item 2
3. Item 3
    + Item 3a
    + Item 3b
```
```

R Code chunks

- R code placed in *chunks* will be evaluated and printed
~~~
```{r}
summary(cars$dist)
summary(cars$speed)
~~~
```
- Inline R Code
```
There were `r nrow(cars)` cars studied
```
- Links: use a plain http address or add a link to a phrase:
```
http://example.com
[linked phrase](http://example.com)
```
- Images on the web or local files in the same directory:
```
![alt text](http://example.com/logo.png)
![alt text](figures/img.png)
```


RStudio
File Edit Code View Plots Session Build Debug Tools Help
report.Rmd*
Knit Word
Knit HTML
Knit PDF
Knit Word
Run
Chunks
Environment History
To Console
To Source
Project: (None)
Files Plots Packages Help Viewer
Zoom
Export
Clear All
8:1
(Top Level)
R Markdown

```
---
title: "Statistics analysis for paper Ludigsa et al."
author: "Frédéric Schütz"
date: "4 November 2014"
output: word_document
---

# Figure 1a (new)

The p-values are obtained using a two-sample (two-sided) t-test. Adjustment for
multiple testing is done using a Bonferroni correction over 12 samples.

```{r echo=FALSE}

dotable <- function( datafile ) {

  data <- read.table(datafile, fill=TRUE, header=FALSE,
                     sep="\t", as.is=TRUE)

  results <- NULL

  for (group in unique(data$V1)) {

    # Create section here !

    for (group2 in unique(data$V2)) {
      for (genotype in c("Nlrc5", "Rfx5")){

        thisdata <- data[ grep( genotype, data$V3 ), ]
```

```
date "4 November 2014"
```

Console ~/sibtmp/guerda_greta/
```
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.

> 
```

Exercises

- Using Rstudio, start a new .Rmd (R Markdown file).
- Look at the template that was provided, change the R code
- Create an HTML, a Word and a PDF file from this Markdown code
- Note: you may need to install a TeX distribution to generate PDF; you can also generate a Word or Excel document, and print/convert them to PDF if required
- Make sure to include information about the current R session (R version, packages loaded) in the final document
- Adapt an R script of your choice (ideally one you would use in your work) in a Markdown report
- Use Git to manage these files.

Generating random numbers on a computer

Using a "real" random number generator (mostly for cryptography)

Generating random numbers for scientific simulations

In scientific simulations, we usually need sequences of **numbers that look random** (that is: looking at a series of number, we can not predict what the next one will be), but that remain **predictable and repeatable** when needed.

Otherwise, debugging is difficult, and it is impossible to verify the results obtained by others.

Using a pseudo-random number generator

A pseudo-random number generator (PRNG) fullfils this task; it usually includes two parts:

- A **seed**: an initial value
- A function that generates a new "random" number based on the previous ones (or on the seed)

Example: the linear congruential method

The series of random numbers is given by

$$X_{n+1} = (a\, X_n + c) \bmod m$$

where

a,c and m are (**well-chosen**) constants;

X_n is the previous random number (or the seed)

X_{n+1} is the next random number

Examples

A good example

$$X_{n+1} = (48271\, X_n + c) \bmod (2^{31} - 1)$$

A bad example:

$$X_{n+1} = (65539\, X_n) \bmod 2^{31}$$

Called RANDU, this generator was used in most of the computers for more than a decade; it actually fails most criteria for randomness !

What happens in R ?

```
Random                  package:base                  R Documentation

Random Number Generation

Description:

   ‘.Random.seed’ is an integer vector, containing the random number
   generator (RNG) *state* for random number generation in R.  It can
   be saved and restored, but should not be altered by the user.

   ‘RNGkind’ is a more friendly interface to query or set the kind of
   RNG in use.

   ‘RNGversion’ can be used to set the random generators as they were
   in an earlier R version (for reproducibility).

   ‘set.seed’ is the recommended way to specify seeds.
```

```
Details:

    The currently available RNG kinds are given below.  ‘kind’ is
    partially matched to this list.  The default is
    ‘"Mersenne-Twister"’.

    ‘"Wichmann-Hill"’ The seed, ‘.Random.seed[-1] == r[1:3]’ is an
         integer vector of length 3, where each ‘r[i]’ is in ‘1:(p[i]
         - 1)’, where ‘p’ is the length 3 vector of primes, ‘p =
         (30269, 30307, 30323)’.  The Wichmann-Hill generator has a
         cycle length of 6.9536e12 (= ‘prod(p-1)/4’, see _Applied
         Statistics_ (1984) *33*, 123 which corrects the original
         article).

    ‘"Marsaglia-Multicarry"’: A _multiply-with-carry_ RNG is used, as
         recommended by George Marsaglia in his post to the mailing
         list ‘sci.stat.math’.  It has a period of more than 2^60 and
         has passed all tests (according to Marsaglia).  The seed is
         two integers (all values allowed).
```

```
‘"Mersenne-Twister"’: From Matsumoto and Nishimura (1998). A
      twisted GFSR with period 2^19937 - 1 and equidistribution in
      623 consecutive dimensions (over the whole period).  The
      ‘seed’ is a 624-dimensional set of 32-bit integers plus a
      current position in that set.

‘"Knuth-TAOCP-2002"’: A 32-bit integer GFSR using lagged Fibonacci
      sequences with subtraction.  That is, the recurrence used is

                   X[j] = (X[j-100] - X[j-37]) mod 2^30

      and the ‘seed’ is the set of the 100 last numbers (actually
      recorded as 101 numbers, the last being a cyclic shift of the
      buffer).  The period is around 2^129.
```

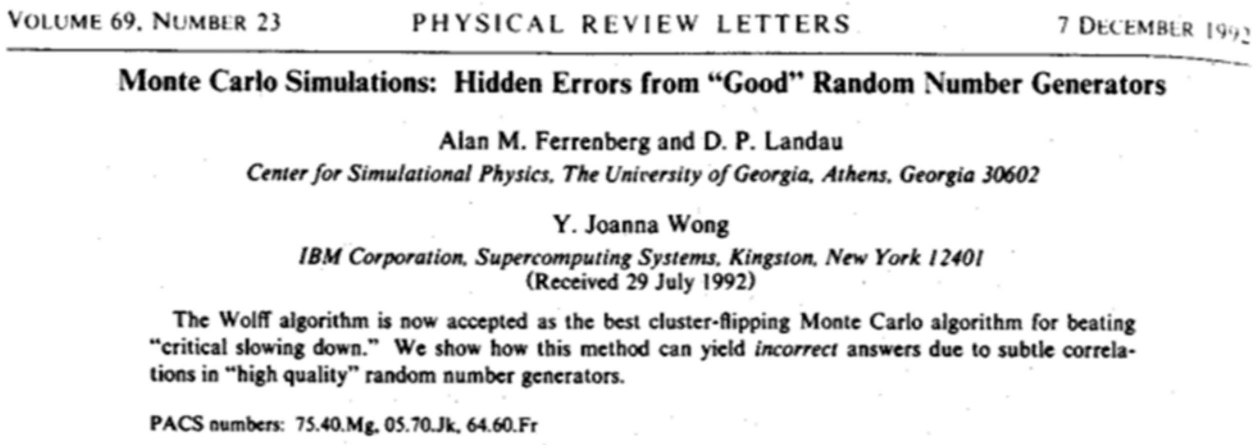
What about the seed ?

The seed should be random if we want random numbers (we need a bit of randomness to start the system, and it will then produce more randomness)

From the R help:

```
Initially, there is no seed; a new one is created from the current
  time (and since R 2.14.0, the process ID) when one is required.
  Hence different sessions will give different simulation results,
  by default.  However, the seed might be restored from a previous
  session if a previously saved workspace is restored.
```

A tale of caution



VOLUME 69, NUMBER 23 PHYSICAL REVIEW LETTERS 7 DECEMBER 1992

Monte Carlo Simulations: Hidden Errors from "Good" Random Number Generators

Alan M. Ferrenberg and D. P. Landau
Center for Simulational Physics, The University of Georgia, Athens, Georgia 30602

Y. Joanna Wong
IBM Corporation, Supercomputing Systems, Kingston, New York 12401

(Received 29 July 1992)

The Wolff algorithm is now accepted as the best cluster-flipping Monte Carlo algorithm for beating "critical slowing down." We show how this method can yield *incorrect* answers due to subtle correlations in "high quality" random number generators.

PACS numbers: 75.40.Mg, 05.70.Jk, 64.60.Fr

Record the seed used when generating random numbers

The set.seed() command allows one to choose a seed, so that the sequence of random numbers can be repeated.

Always record that seed, so that the results can be reproduced.

Suggestions:

`set.seed(1)`

for exercices

`set.seed(201404041)`

for real simulations
(reproducible, easy, and not duplicated)

```
# Generate a dataset
set.seed(1)
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
data <- data.frame(x, y)

# Fit a linear model
model <- lm( data$y ~ data$x )

# Generate a second dataset
x <- runif(100)
y <- 2*x + rnorm(length(x))/10
newdata <- data.frame(x, y)

# Use the linear model to perform a prediction on the newdata
predict(model, newdata)
```

This code does not return any error message, but it does not work. Why ?