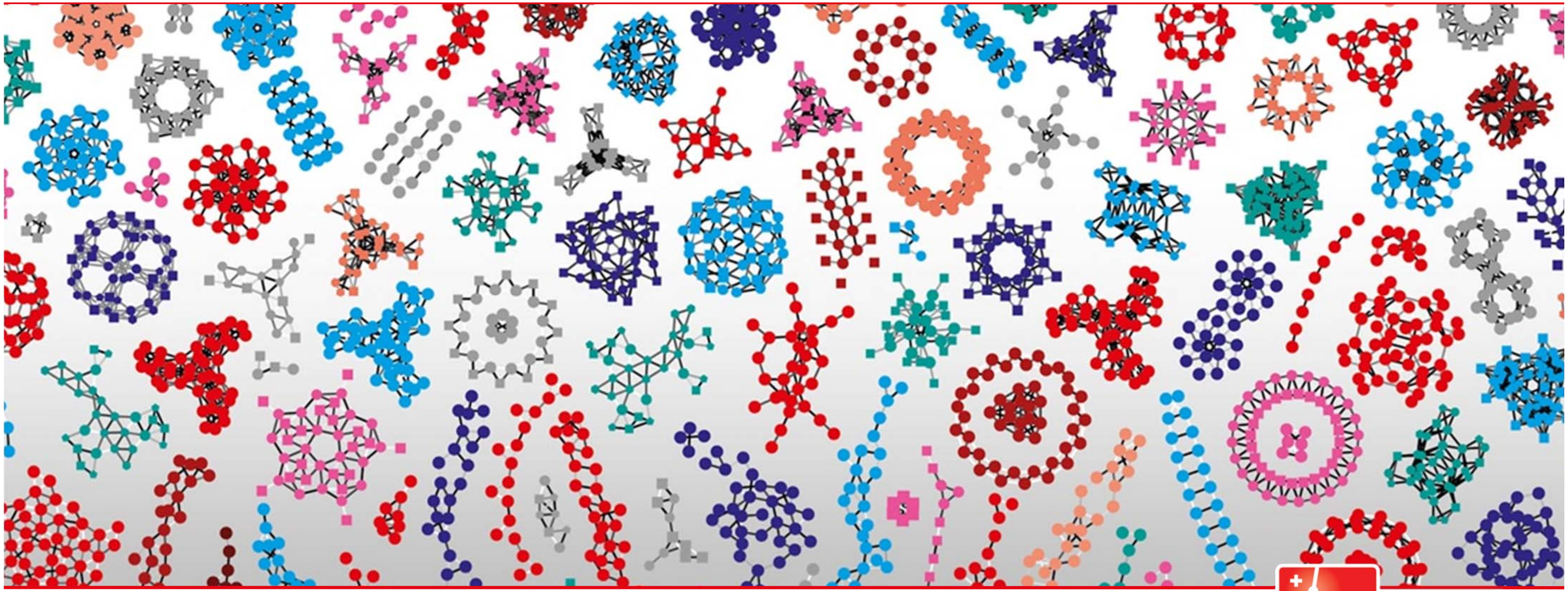


Swiss Institute of Bioinformatics

Advanced R: Namespaces and profiling

Frédéric Schütz (Frederic.Schutz@sib.swiss)

Namespaces

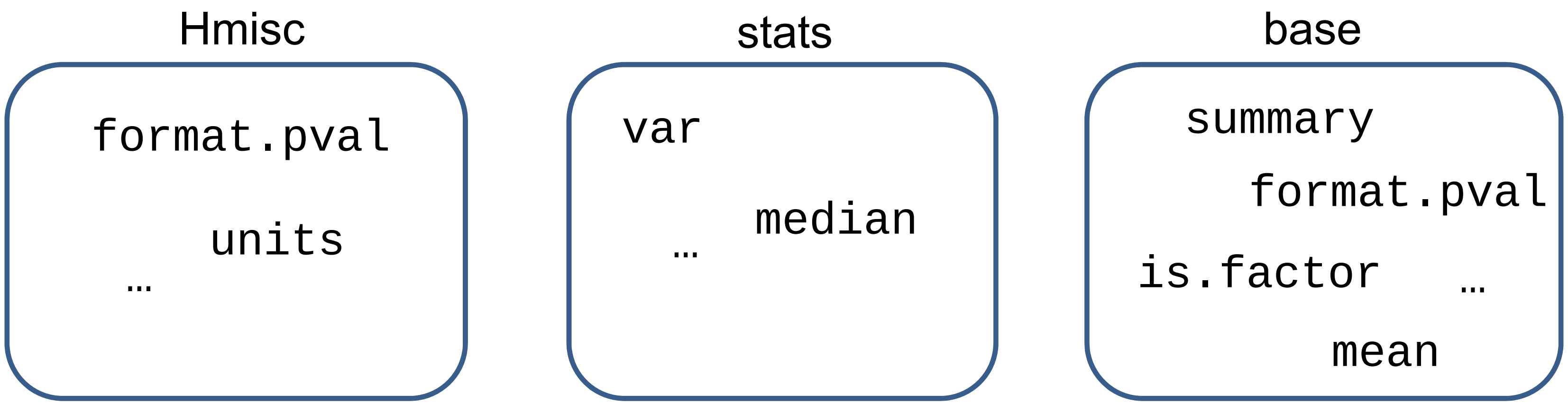
What happens when several packages define the same function?

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
```

Namespaces

Each R package has its own **namespace** for objects, so that several packages can provide the same function without any interference.



Hmisc

`format.pval`

`units`

…

stats

`var`

`median`

…

base

`summary`

`format.pval`

`is.factor`

…

`mean`

Search path

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"
```

Namespaces

Functions from different packages can be differentiated using `::`

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

Namespaces

If you display the code of the function from a package, the namespace is printed afterwards

```
> Hmisc::format.pval
function (x, pv = x, digits = max(1, .Options$digits - 2),
          eps = .Machine$double.eps, na.form = "NA", ...)
{
    if ((has.na <- any(ina <- is.na(pv))))
    ...
    r
}
<bytecode: 0x55be43c290d0>
<environment: namespace:Hmisc>
```

Namespaces

If you display the code of the function from a package, the namespace is printed afterwards

```
> base::format.pval
function (x, pv = x, digits = max(1, .Options$digits - 2),
          eps = .Machine$double.eps, na.form = "NA", ...)
{
    if ((has.na <- any(ina <- is.na(pv))))
    ...
    r
}
<bytecode: 0x55be43c290d0>
<environment: namespace:base>
```

Namespaces

This allows the redefinition of a function, while 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
  …
}
```

After deleting the new function, the original one remains available.

Be careful to avoid recursion

Make sure the function does not call itself !

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

  # Then we modify the output as we want
  …
}
> summary(1:10)
Error: C stack usage  7970068 is too close to the limit
```

Example

```
`+` <- function(x, y) {
         base::`+`( base::`+`(x, y), 0.1 )
       }
```

Example: redefining the addition

```
> 1+1
[1] 2
> `+` <- function(x, y) { base::`+`( base::`+`(x, y), 0.1 )  }
> 1+1
[1] 2.1
> rm(`+`)   # Go back to a sane version of the addition.
> 1+1
[1] 2
```

Exporting functions

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` (available from outside) but not `t.test.default`, which you are supposed to call through the generic function `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, there is usually a good reason for the function not to be exported

getAnywhere: finds all namespaces containing a given function

```
> library(Hmisc)
> getAnywhere(format.pval)
2 differing objects matching ‘format.pval’ were found
in the following places
  package:Hmisc
  package:base
  registered S3 method for format from namespace Hmisc
  namespace:base
  namespace:Hmisc
Use [] to view one of them
> getAnywhere(format.pval)[1]
function (pv, digits = max(1L, getOption("digits") - 2L),
...
<environment: namespace:Hmisc>
```

Remember…

How does R store both the `c()` function and the c vector, and how does it differentiate between them ?

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

The vector and the function belong to different namespaces

```
> c=c(c=c)
> c=c(c="c")
> getAnywhere(c)
2 differing objects matching ‘c’ were found
in the following places
  .GlobalEnv
  package:base
  namespace:base
Use [] to view one of them
```

If you try to run a function, R will ignore all other types of variables it will find

Environments

An environment is a data structure that contains R objects.

Each package or function has its own environment, which defines the variables it has access to.

Each environment also has a parent environment; variables in the parent environment are also available to the function.

```
> environment()
<environment: R_GlobalEnv>
> f <- function() { environment() }
> f()
<environment: 0x55b02effa7e8>
```

Lexical scoping

```
> a
Error: object 'a' not found
> f <- function() { a <- 1 }
> f()
> a
Error: object 'a' not found
> a <- 2
> f
function() { a <- 1 }
> a
[1] 2
> f <- function() { print(a) }
> f()
[1] 2
> f <- function() { a <- 1; print(a) }
> a
[1] 2
> f()
[1] 1
> a
[1] 2
```

<-

vs

=

Assignment operators in R

R provides 5 assignment operators:

```
?assignOps

    Description
    Assign a value to a name.

    Usage
    x <- value
    x <<- value
    value -> x
    value ->> x
    x = value
```

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 Wickham'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)    # work
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.

Do you use
the `attach()` command ?

The `attach` *command adds a dataframe or list into the search path*

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

The rule is simple:

Never use `attach()`

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
```

What happens with the search path

```
> data <- list(a=1, b=2)
> search()
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
> attach(data)
> search()
 [1] ".GlobalEnv"        "data"              "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

What happens with the search path

```
> attach(data)
The following objects are masked from data (pos = 3):

    a, b

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

Attached data remains even after deleting everything

```
> rm(list=ls())
> ls()
character(0)


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

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()`.

One more question…

What does the `library()` command do ?

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 ) )
    ...
} )
```

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
```

Can we improve the code further ?

```
n <- 100000
m <- 100

results <- vector("numeric", n)

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

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

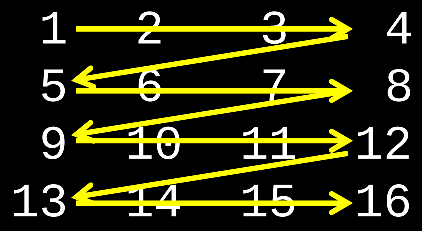
```
 system.time(
 for (i in 1:n) {
     result <- mean( runif( m ) )
 } )

                                    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)

# 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]
  }
}
```

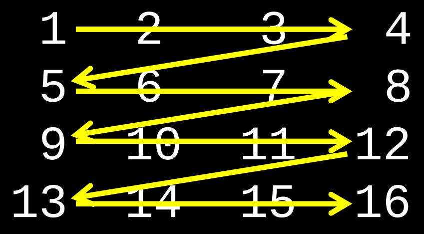
Is there any practical difference between these two loops ?

```
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]
  }
}
```

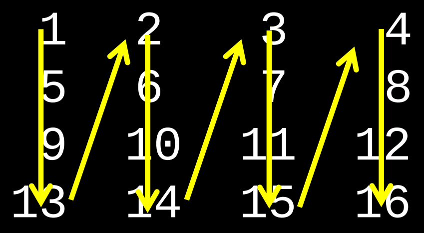


Is there any practical difference between these two loops ?

```
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]
  }
}
```



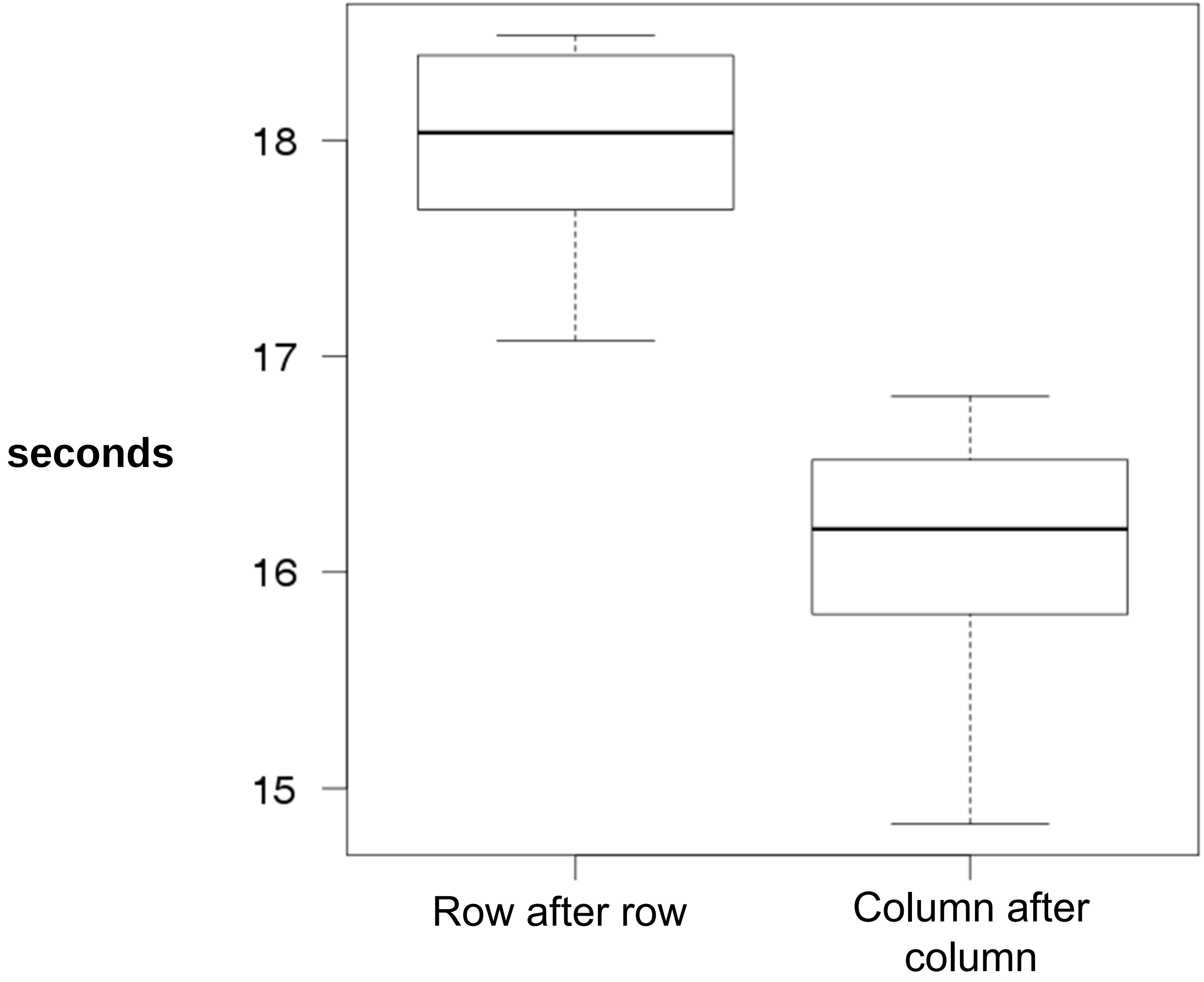
Is there any practical difference between these two loops ?

```
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 anecdotes 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
...
```

Displaying the results of the profiling

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

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
...
```

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
...
```

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

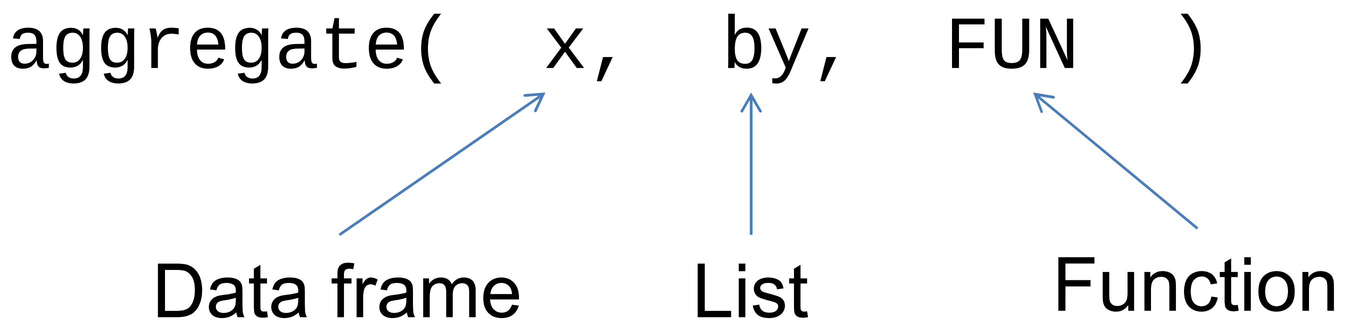
```
> head(data)
  sex height
1   M    183
2   M    183
3   M    182
4   M    175
5   M    158
6   M    179
> tapply(data$height, data$sex, FUN=mean)
       F        M
166.1739 178.2500
```

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
```

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)