Rcourse:
Programming in R

Sonja Grath, Noémie Becker & Dirk Metzler

Winter semester 2014-15

1. Back to input files
   - Review on data frame
   - Factors

2. Programming
   - Conditional execution
   - Loops
   - Executing a command from a script
   - Writing your own functions
   - lapply() and tapply()
   - How to avoid slow R code

Contents

Review on data frame

Generic functions:

```
> read.table()
> write.table()
```

Review on data frame

Generic functions:

```
> read.table()
> write.table()
```

Example 1: wghtcls ”smoker” lifespan
”3” 0 50.3
3 0 52.8

Review on data frame

Generic functions:

```
> read.table()
> write.table()
```

Example 1: wghtcls ”smoker” lifespan
”3” 0 50.3
3 0 52.8

```
> riscfactor <-
read.table("lifespandata2.txt",header=TRUE)
```

Review on data frame

Example 2: wghtcls,smoker,lifespan
3,0,50.3
3,0,52.8

Review on data frame

Example 2: wghtcls,smoker,lifespan
3,0,50.3
3,0,52.8

```
> riscfactor <- read.csv("lifespandata.csv")
> riscfactor <-
read.table("lifespandata.csv",header=TRUE, sep=",",
fill=TRUE)
```

Review on data frame

Example 2: wghtcls,smoker,lifespan
3,0,50.3
3,0,52.8

```
> riscfactor <- read.csv("lifespandata.csv")
> riscfactor <-
read.table("lifespandata.csv",header=TRUE, sep=",",
fill=TRUE)
```

Example 3: weight class smoker lifespan
3 0 50.3
3 0 52.8

```
> riscfactor <-
read.table("lifespandata2.txt",header=TRUE)
```

Review on data frame

Example 2: wghtcls,smoker,lifespan
3,0,50.3
3,0,52.8

```
> riscfactor <- read.csv("lifespandata.csv")
> riscfactor <-
read.table("lifespandata.csv",header=TRUE, sep=",",
fill=TRUE)
```

Example 3: weight class smoker lifespan
3 0 50.3
3 0 52.8

```
> riscfactor <-
read.table("lifespandata2.txt",header=TRUE)
```

You have to change the first line of the file because of the space between weight and class.

Contents

Factors

A variable (numeric or text) can be intended as a factor.

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
> str(x)
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
> str(x)
chr [1:5] "female" "male" "male" "female" "female"
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
> str(x)
chr [1:5] "female" "male" "male" "female" "female"
> x <-factor(x)
```

Factors

A variable (numeric or text) can be intended as a factor.

Example with text:

```
> x <- c("female","male","male","female","female")
> levels(x)
NULL
> str(x)
chr [1:5] "female" "male" "male" "female" "female"
> x <-factor(x)
> levels(x)
[1] "female" "male"
> str(x)
Factor w/ 2 levels "female","male":  1 2 2 1 1
```

Factors

Example with numbers:

```
> y <- rep(c(17,17,18),4); str(y)
num [1:12] 17 17 18 17 17 18 17 17 18 17 ...
```

Factors

Example with numbers:

```
> y <- rep(c(17,17,18),4); str(y)
num [1:12] 17 17 18 17 17 18 17 17 18 17 ...
> summary(y)
Min.  1st Qu.  Median Mean 3rd Qu.  Max.
17.00 17.00 17.00 17.33 18.00 18.00
```

Factors

Example with numbers:

```
> y <- rep(c(17,17,18),4); str(y)
num [1:12] 17 17 18 17 17 18 17 17 18 17 ...
> summary(y)
Min.  1st Qu.  Median Mean 3rd Qu.  Max.
17.00 17.00 17.00 17.33 18.00 18.00
> y <- factor(y); str(y)
Factor w/ 2 levels "17","18":  1 1 2 1 1 2 1 1 2 1 ...
> summary(y)
17 18
8 4
```

Back to input files

By default read.table() sets text variables as factors and not numerical variables.

Back to input files

By default read.table() sets text variables as factors and not numerical variables.
This can be changed by specifying the class of the columns.

```
riscfactor <-
read.table("lifespandata.txt",header=TRUE,
colClasses=c("factor","numeric","numeric"))
```

Back to input files

By default read.table() sets text variables as factors and not numerical variables.
This can be changed by specifying the class of the columns.

```
riscfactor <-
read.table("lifespandata.txt",header=TRUE,
colClasses=c("factor","numeric","numeric"))
```

Or by changing the variables afterwards.

```
riscfactor$wghtcls <- factor(riscfactor$wghtcls)
```

Contents

Conditional execution

If(), else() and ifelse()

Conditional execution

If(), else() and ifelse()
Syntax:

```
if ( condition ) { commands1 }
if ( condition ) { commands1 } else { commands2 }
ifelse ( conditions vector, yes vector, no vector )
```

Conditional execution

If(), else() and ifelse()
Syntax:

```
if ( condition ) { commands1 }
if ( condition ) { commands1 } else { commands2 }
ifelse ( conditions vector, yes vector, no vector )
```

Example:

```
> x <- 4
> if (x==5) {x <- x+1} else {x <- x*2}
```

Conditional execution

If(), else() and ifelse()
Syntax:

```
if ( condition ) { commands1 }
if ( condition ) { commands1 } else { commands2 }
ifelse ( conditions vector, yes vector, no vector )
```

Example:

```
> x <- 4
> if (x==5) {x <- x+1} else {x <- x*2}
> x
[1] 8
```

Conditional execution

Other examples:

```
x <- 8
if ( x != 5 & x>3 ) { x <- x+1 ; 17+2 } else { x <- x*2
; 21+5 }
```

Conditional execution

Other examples:

```
x <- 8
if ( x != 5 & x>3 ) { x <- x+1 ; 17+2 } else { x <- x*2
; 21+5 }
[1] 19
> x
[1] 9
```

Conditional execution

Other examples:

```
x <- 8
if ( x != 5 & x>3 ) { x <- x+1 ; 17+2 } else { x <- x*2
; 21+5 }
[1] 19
> x
[1] 9

> y <- 1:10
> ifelse( y<6, y^2, y-1 )
```

Conditional execution

Other examples:

```
x <- 8
if ( x != 5 & x>3 ) { x <- x+1 ; 17+2 } else { x <- x*2
; 21+5 }
[1] 19
> x
[1] 9

> y <- 1:10
> ifelse( y<6, y^2, y-1 )
[1] 1 4 9 16 25 5 6 7 8 9
```

Contents

Loops

For(), while() and repeat()

Loops

For(), while() and repeat()
Syntax:

```
for ( var in set ) { commands }
while ( condition ) { commands }
repeat { commands }
```

Loops

For(), while() and repeat()
Syntax:

```
for ( var in set ) { commands }
while ( condition ) { commands }
repeat { commands }
```

`break` stops all loops
`next` goes directly to the next iteration of the loop

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
```

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
 # i=3 is skipped, so x <- 1+2+4+5
> x
[1] 12
```

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
 # i=3 is skipped, so x <- 1+2+4+5
> x
[1] 12

> y <- 1; j <- 1
> while ( y < 12 & j < 8 ) { y <- y*2 ; j <- j + 1}
```

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
 # i=3 is skipped, so x <- 1+2+4+5
> x
[1] 12

> y <- 1; j <- 1
> while ( y < 12 & j < 8 ) { y <- y*2 ; j <- j + 1}
```

y is 16 and j is 5

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
 # i=3 is skipped, so x <- 1+2+4+5
> x
[1] 12

> y <- 1; j <- 1
> while ( y < 12 & j < 8 ) { y <- y*2 ; j <- j + 1}
```

y is 16 and j is 5

```
> z <- 3
> repeat { z<- z^2; if ( z>100 ) { break }; print(z)}
```

Examples

```
> x <- 0
> for ( i in 1:5 ) { if (i==3) { next } ; x <- x + i }
 # i=3 is skipped, so x <- 1+2+4+5
> x
[1] 12
```

```
> y <- 1; j <- 1
> while ( y < 12 & j < 8 ) { y <- y*2 ; j <- j + 1}
```

y is 16 and j is 5

```
> z <- 3
> repeat { z<- z^2; if ( z>100 ) { break }; print(z)}
[1] 9
[1] 81
```

The loop stopped after 81^2 so z is 6561.

Contents

Executing a command from a script

R scripts and stored in .R or .r files and are executed with the command `source()`

Executing a command from a script

R scripts and stored in .R or .r files and are executed with the command `source()`

```
source(C:/Documents/R/myscript.R)
```

Executing a command from a script

R scripts and stored in .R or .r files and are executed with the command `source()`

```
source(C:/Documents/R/myscript.R)
```

You can specify the current working directory using the command `setwd()`

Executing a command from a script

R scripts and stored in .R or .r files and are executed with the command `source()`

```
source(C:/Documents/R/myscript.R)
```

You can specify the current working directory using the command `setwd()`

```
setwd(C:/Documents/R)
getwd()
```

Contents

Basics

Syntax:

```
myfun <- function (arg1, arg2, .   .   .)  { commands }
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, .   .   .)  { commands }
```

Examples:

```
se <- function(x)
{
y<-sqrt(var(x)/length(x))
return(y)
}
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, .   .   .)  { commands }
```

Examples:

```
se <- function(x)
{
y<-sqrt(var(x)/length(x))
return(y)
}
se(1:4)
[1] 0.6454972
```

Basics

Syntax:

```
myfun <- function (arg1, arg2, .   .   .)  { commands }
```

Examples:

```
se <- function(x)
{
y<-sqrt(var(x)/length(x))
return(y)
}
se(1:4)
[1] 0.6454972
 se("wrong type of argument")
[1] NA
Warning message:
In var(x) :  NAs introduced by coercion
```

Deal with non correct arguments

Add to the previous function (before the formula for y):

```
if (is.numeric(x)!=TRUE)
{
stop("need numeric data")
}
```

Deal with non correct arguments

Add to the previous function (before the formula for y):

```
if (is.numeric(x)!=TRUE)
{
stop("need numeric data")
}
```

```
 se("wrong type of argument")
Error in se("wrong type of argument") :  need numeric
data
```

Deal with the missing data

By default, many R fucntions erase missing data automatically. You can do the same by adding to your function:

```
x<-x[is.na(x)==FALSE]
```

Deal with the missing data

By default, many R fucntions erase missing data automatically. You can do the same by adding to your function:

```
x<-x[is.na(x)==FALSE]
```

```
 se(c(1:4,NA))
[1] 0.6454972
```

Add other arguments

R functions can have several arguments. Here for example you could define an argument to control whether R should remove the NA values or not (this is what is implemented in many R fucntions):

```
se <- function(x,na.rm=FALSE)
False is the default value of the argument na.rm (more
about this next slide).  { if (is.numeric(x)!=TRUE)
{stop("need numeric data")}
if (na.rm==TRUE){x<-x[is.na(x)==FALSE]}
y<-sqrt(var(x)/length(x))}
```

Add other arguments

R functions can have several arguments. Here for example you could define an argument to control whether R should remove the NA values or not (this is what is implemented in many R fucntions):

```
se <- function(x,na.rm=FALSE)
False is the default value of the argument na.rm (more
about this next slide).  { if (is.numeric(x)!=TRUE)
{stop("need numeric data")}
if (na.rm==TRUE){x<-x[is.na(x)==FALSE]}
y<-sqrt(var(x)/length(x))}
```

You can omit to write the name of the argument:

```
 se(c(NA,1:4), TRUE))
[1] 0.6454972
```

Add other arguments

R functions can have several arguments. Here for example you could define an argument to control whether R should remove the NA values or not (this is what is implemented in many R fucntions):

```
se <- function(x,na.rm=FALSE)
False is the default value of the argument na.rm (more
about this next slide).  { if (is.numeric(x)!=TRUE)
{stop("need numeric data")}
if (na.rm==TRUE){x<-x[is.na(x)==FALSE]}
y<-sqrt(var(x)/length(x))}
```

You can omit to write the name of the argument:

```
 se(c(NA,1:4), TRUE))
[1] 0.6454972
```

Or give na.rm before the vector.

But not both (omit name and change order of arguments).

Giving default values to the arguments

Imagine a function multiplies a number by a predefined other number.

```
mymul <- function(x, n){
return(x*n)}
```

Giving default values to the arguments

Imagine a function multiplies a number by a predefined other number.

```
mymul <- function(x, n){
return(x*n)}
```

You can set by default the value of n to 2:

```
mymul <- function(x, n=2){
return(x*n)}
```

Giving default values to the arguments

Imagine a function multiplies a number by a predefined other number.

```
mymul <- function(x, n){
return(x*n)}
```

You can set by default the value of n to 2:

```
mymul <- function(x, n=2){
return(x*n)}
```

`mymul(2)` gives as answer 4

`mymul(2,3)` becomes 6

Returning several values

To do so use a vector or a list.

Returning several values

To do so use a vector or a list.

```
ci.norm <- function(x,conf=0.95)
{
q <- qnorm(1-(1-conf)/2)
return(
list(lower=mean(x)-q*se(x),upper=mean(x)+q*se(x)))
}
```

Returning several values

To do so use a vector or a list.

```
ci.norm <- function(x,conf=0.95)
{
q <- qnorm(1-(1-conf)/2)
return(
list(lower=mean(x)-q*se(x),upper=mean(x)+q*se(x)))
}

ci.norm(rnorm(100))
$lower [1] -0.1499551
$upper [1] 0.2754680

ci.norm(rnorm(100,conf=0.99))
$lower [1] -0.1673693
$upper [1] 0.2443276
```

Contents

lapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

lapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

tapply() is used for data frames.

lapply() and tapply()

You use apply() and its derivatives to apply the same function to each element of an object.

```
v <- 1:4
sapply(v,factorial)
# returns a vector, lapply() would return a list
[1] 1 2 6 24
```

tapply() is used for data frames.
Example: data frame containing lifespan for people from 3 classes of weight. You want the mean lifespan for each class.

```
tapply(lifespan,weightcls,mean)
1 2 3
69 61 53
```

Contents

How to avoid slow R code

- R has to interpret your commands each time you run a script and it takes time to determine the type of your variables.
- So avoid using loops and calling functions again and again if possible
- When you use loops, avoid increasing the size of an object (vector ..:) at each iteration but rather define it with full size before.
- Think in whole objects such as vectors or lists and apply operations to the whole object instead of looping through all elements.