Programming with R

functions and loops

Emmanuel Kemel (HEC Paris & CNRS)

Notions seen in these slides

- user-defined functions
- loops and implicit loops
- class-oriented programming

Outline

User-defined functions

- R allows to write your own functions
- This is the principle of packages: they load scripts of user-defined functions
- There is a specific mode for functions.

The function class

```
max(1:10)
mode(max)
max=1
mode(max)
max(1:10)
```

Why writing your own functions?

- Because the function does not exist
- In order to customize existing functions to your particular needs
  For example, **lm()** does not return
  - scatter plots
  - heterosdecasticity-robust standard errors

  You may thus want to adapt the function so that it returns these elements.
- Because some functions take user defined function as arguments (e.g. **optim()**)

Example: comparing two variables

- Suppose you need to run a series of statistics in order to compare two variables x_1 and x_2,
  more precisely, you want: means, medians, sd, IQR, t.tests, wilcoxon test, plots the boxplots.

```
setwd("~/Dropbox/Rprogramming/data")
data=read.table("cps08.csv", header=TRUE, sep=";", dec=",")
data$gender=factor(data$female,labels=c("M","F"))
data$bachelor=factor(data$bachelor,labels=c("No Bachelor","Bachelor

tab=aggregate(ahe~gender,data=data, FUN=mean)
t=t.test(ahe~gender,data=data)
w=wilcox.test(ahe~gender,data=data)
plot(ahe~gender,data=data)
c(tab$ahe, t$p.value, w$p.value)
```

Example: comparing two variables (continued)

```
my_comparison=function(formula, data){
formula=as.formula(formula)
tab=aggregate(formula,data=data, FUN=mean)
t=t.test(formula,data=data)
w=wilcox.test(formula,data=data)
plot(formula,data=data)
return(c(tab[,2], t$p.value, w$p.value))
}

my_comparison("ahe~bachelor", data)
my_comparison("ahe~bachelor", data[data$gender=="F",])
```

The general syntax

The general syntax for the definition of a function is
function_name=function(arg1=value_by_default1,
arg2=value_by_default2, ...**){**
some expression
return()
}

- Parts in bold are mandatory, other parts are optional
- If you choose the name of an existing (even built-in) function, it will be replaced.
- the script of a user-written function can be printed with print()

arguments are identified by their position or by their name

Simple examples

- default arguments and return

```
geo.mean=function(x){prod(x)^(1/length(x))}
geo.mean(1:10)
```

```
[1] 4.528729
```

```
geo.mean()
```

```
Error in geo.mean(): l'argument "x" est manquant, avec aucune valeur par d'efaut
```

```
geo.mean=function(x=1){prod(x)^(1/length(x))}
geo.mean()
```

```
[1] 1
```

```
geo.mean=function(x=1){
prod=prod(x)
power=1/length(x)
result=prod^power
return(result)
}
```

Without return, the value of the last evaluated expression is returned automatically in R.

Local versus global variables

- Functions use in priority local variables when possible, global variables otherwise.
- Assignments made in function are local, if not explicitly told otherwise.

```
x=1
test1=function(x){print(x)}
test1(5)
```

```
[1] 5
```

```
test2=function(y){x=y; print(x)}
test2(5)
```

```
[1] 5
```

```
print(x)
```

```
[1] 1
```

```
x=1
test2=function(y){x<<-y; print(x)}
test2(5)
```

```
[1] 5
```

```
print(x)
```

```
[1] 5
```

Other optional arguments

- Room can be left for other non-specified arguments using "..."

```
x=runif(100)
some_plot=function(x, fun){plot(fun(x))}
some_plot(x,sqrt)
some_plot(x,sqrt, main="Random values")
some_plot=function(x, fun,...){plot(fun(x),...)}
some_plot(x,sqrt, main="Random values")
```

My slm

- Create a function slm() that
  - takes two arguments: the dependent variables and the regressor
  - computes
    - OLS estimators
    - computes the R^2
    - the residuals
  - returns a list that contains these results and
  - plots a scatter plot and the estimated regression line

Solution

```
slm=function(y,x){
b1=cov(x,y)/(var(x))
b0=mean(y)-b1*mean(x)
haty=b0+b1*x
e=y-haty
r2=var(haty)/var(y)
result=list(coefficient=c(b0,b1),residuals=e,rsquared=r2 )
plot(y~x)
abline(a=b0,b=b1, col="red")
return(result) }
```

Other example

- the following function defines a "robust summary" of an lm object that replaces defaults standard errors by heteroscedasticity robust standard errors.

```
robsummary=function(mod){
library(sandwich)
vcov = vcovHC(mod, "HC1")
se=sqrt(diag(vcov))
result=summary(mod)
result$coefficients=cbind(mod$coefficient,se,mod$coefficient-1.96*se,mo
colnames(result$coefficients)=c("coefficients","rob_se","rob95CI_inf","
return(result) }
```

Recursion

Recursive functions can be defined easily:

- the function can be called within the definition of the (same) function
- ex: create a function that compute $!n$

Control structures

- In your functions, you may need control structures in order to
  - check that the arguments have the proper form
  - condition the calculations on the values of the arguments
  - specify initial values in recursive programming
- In order to make an operation conditional to a condition, the following syntax will be used.

```
if (condition) {
expression }
```

```
if (condition)
{expression1}
else {expression2}
```

Outline

Why using loops?

Loops are used to automatize one (or several) series of operations.

- Examples:
  - repeat a statistical treatment on several datasets/units
  - systematically scan values of an R object
  - iterate an operation until a condition is satisfied (numerical maximization)
  - run simulations

"for" loops

With for loops:

- a local variable scans a series of values (in a vector or a list)
- for each value taken by the local variable, a specified operation (that may depend on the current value of the local variable) is run.

With these loops, the total number of time the operation is called is known in advance.

The general syntax is:
for (local_variable_names in vector) {
expression
}

Very simple examples

```
i=10
for (i in 1:5) print(i); print(i)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 5
```

```
i=10
for (i in 1:5) {print("the current value is "
```

```
[1] "the current value is "
[1] 1
[1] "the current value is "
[1] 2
[1] "the current value is "
[1] 3
[1] "the current value is "
[1] 4
[1] "the current value is "
[1] 5
```

A simple exercise

- Write a "for" loop that computes the sum of a vector x

Solution

```
sumx=0
for (i in 1:length(x)) {sumx=sumx+x[i]}
#or
for (i in x) {sumx=sumx+i}
```

Control structures

- Loops become (more) interesting when the operation that is realized for each value of the local variable, depends on this value.
- In order to make an operation conditional to a condition, the following syntax will be used.

if *(condition) {*
expression }

if *(condition)*
{expression1}
else *{expression2}*

Example with control structures

Consider a vector x, print the values of x that are larger than 5.

```
x=1:10
for (i in x){
        if (x>5) {print(x)}
}
```

Simple examples: loops versus vectorization

Consider a vector x with NAs. Write a loop that replaced NAs by the mean of x.

- the loop script is

```
x=rnorm(10); x[c(2,5,7)]=NA
m=mean(x, na.rm=TRUE)
for (i in 1:length(x)){
        if (is.na(x[i])){x[i]=m}
}
```

- the vector structure of R allows to write more easily

```
x=rnorm(10); x[c(2,5,7)]=NA
x[is.na(x)]<-mean(x, na.rm=TRUE)
```

Using vectorized functions allows to write more elegant and faster code.

loops versus vectorization: efficiency

```
x=rnorm(10^6); x[c(2,5,7)]=NA
m=mean(x, na.rm=TRUE)
##############################
system.time(for (i in 1:length(x)){if (is.na(x[i])){x[i]=m}})
   user  system elapsed
  0.145   0.007   0.152
##############################
system.time(x[is.na(x)]<-mean(x, na.rm=TRUE))
   user  system elapsed
  0.014   0.003   0.017
```

Example of useful loops

- The following loop loads and concatenates several data sets into a single data.frame.

```
setwd("~/Dropbox/Rprogramming/data")
files=list.files(pattern="result")
data=c()
for (i in files){data=rbind(data, read.table(i, header=T,sep=";"))}
```

Examples of useful loops (continued)

- The following loop converts a series of Stata (.dta) files into text (.csv) files.

```
library(foreign)
setwd("~/Dropbox/Rprogramming/data/mus_data")
stata_files=list.files(pattern=".dta")
for (file in stata_files){
temp=read.dta(file)
name=sub(".dta",".csv",file)
write.table(temp, file=name)
}
```

Examples of useful loops (continued)

Give summary of each of the treated datasets

```
sink("summary.txt")
txt_files=list.files(pattern=".csv")
for (file in txt_files){
print("############################")
print(file)
print(summary(read.table(file)))
}
sink()
```

Examples of useful loops: graphs

We consider the data set CPS08 and want to represent the gender gap for each age, ranging from 25 to 34, distinguishing people with bachelor degree from people without.

```
setwd("~/Dropbox/Rprogramming/data")
data=read.table("CPS08.csv",header=T, sep=";", dec=",")
data$gender=factor(data$female, labels=c("M","F"))
pdf("gender_gap_per_age_and_education.pdf")
par(mfrow=c(1,2))
for (i in 25:34){
for (j in 0:1){
plot(ahe~gender, data=data, subset=c(age==i & bachelor==j),
main=paste("Gender gap for age ",i))
}}
dev.off()
```

Recommendations when writing loops

- when debugging, place some prints() in the code to follow the process
- use indentation to increase the readability of the code
- use the functionalities offered by RStudio

Implicit loops

Several functions apply operations sequentially to each value of a vector (or matrix or list)
They can be used to avoid writing a loop.

- apply applies expression to each lines or columns of a matrix
- tapply applies expressions to a vector conditionally on the value of another vector
- lapply applies expression to each values of a list
- sapply generalizes lapply (by also applying operations to vectors)
- mapply is a multivariate expression of sapply

Note that user-defined function can be written as arguments of apply-like functions.

Example

```
sapply(list.files(pattern=".dta"),
function(x) write.table(read.dta(x), file=sub(".dta",".csv",x)))
```

Instead of

```
for (file in stata_files){
temp=read.dta(file)
name=sub(".dta",".csv",file)
write.table(temp, file=name)
}
```

While

- "while" loops iterate operations as long as a condition is satisfied and stop when the condition is satisfied.
- There are generally used for numeric search: the search continues until a condition is (not) satisfied.
- The syntax for while loops is
  while (condition) {operation that can change the condition}

They are generally used to built loops for which the number of operations to run is not known in advance.

Example 1

Using a while loop to mimic a for loop.

```
k=1
while (k<=10){print(k); k=k+1}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Example 2

Write a loop that creates a vector containing the n first prime numbers.

```
n=10
result=c()
k=1
while (length(result)<n){
if (sum(k%%1:k==0)==2){result=c(result, k)}
k=k+1
}
print(result); print(k)
```

```
 [1]  2  3  5  7 11 13 17 19 23 29
[1] 30
```

Exercise

Run simulations to illustrate that

1. the distribution of sample means is normal (take a sample size of 50, and 1000 simulations)
2. the variance of sample means vary linearly with sample size (make sample size range from 1 to 100, take 1000 simulations)

Do this for the mean

- a normal distribution
- a binomial distribution of probability 0.5, then 0.2

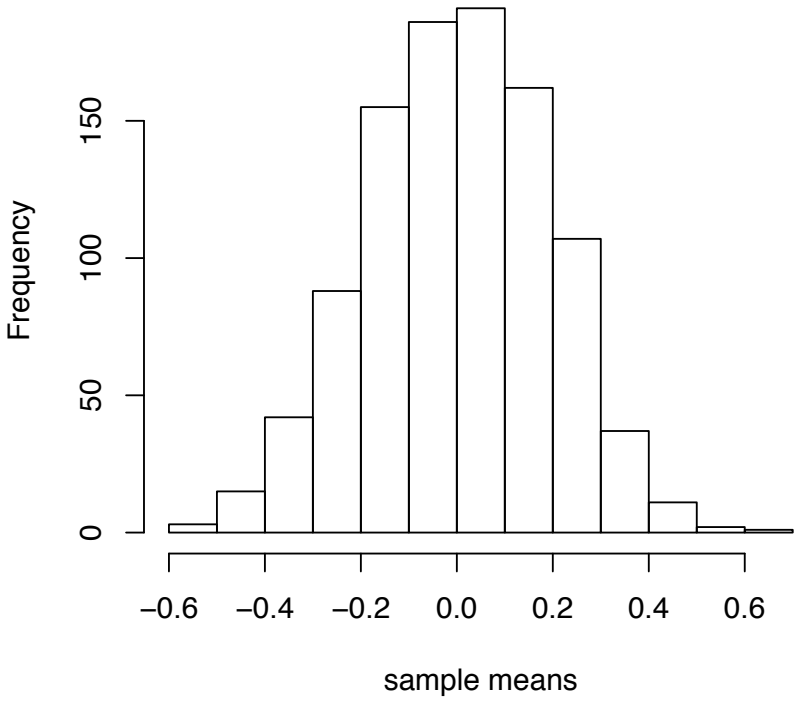
Solution: code

```
n=30; s=1000;
sample_means=c()
for (i in 1:s){
samplei=rnorm(n)
sample_means=c(sample_means,mean(samplei))
hist(sample_means)
```

A more elegant solution

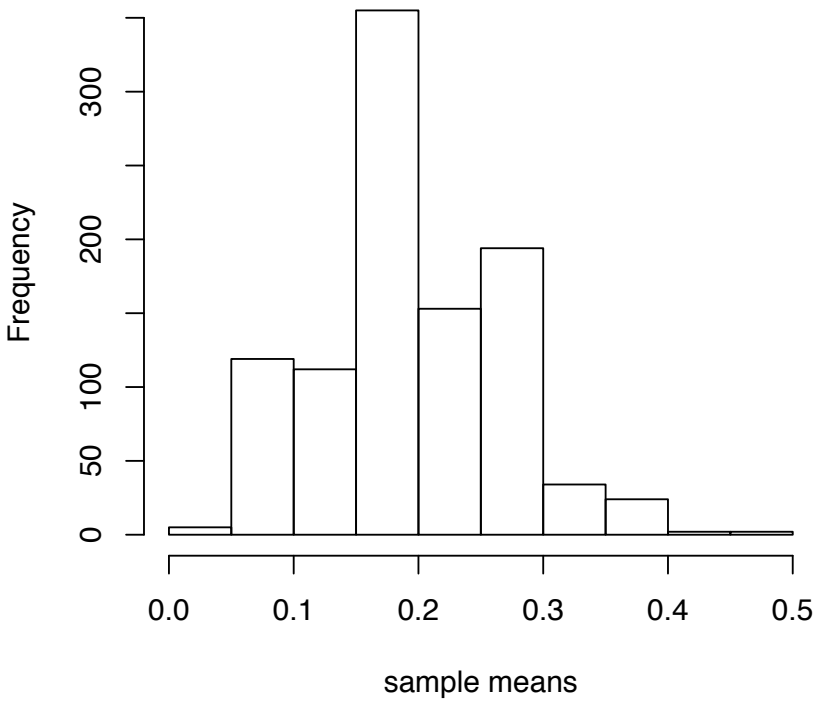
```
hist(sapply(1:s,function(x) mean(rnorm(n))))
```

Solutions: output



Solutions: code and output with binomial distributions

Illustration for a binomial distribution with $p = 0.2$



Solutions: std of sample mean and sample size

Illustration for a binomial distribution with $p = 0.1$

```
smv=c()
for (n in 1:50){
        sample_means=c()
        for (simulation in 1:1000){
                sample=rnorm(n)
                sample_means=c(sample_means,mean(sample))
        }
        smv=c(smv,var(sample_means))
}
plot(1:50, smv, type="p")
curve(1/x,1,50,add=T)
```

A more efficient code

```
smv=sapply(1:50, function(x) var(replicate(1000, mean(rnorm(x)))))
plot(1:50, smv, type="p")
curve(1/x,1,50,add=T)
```

Illustration

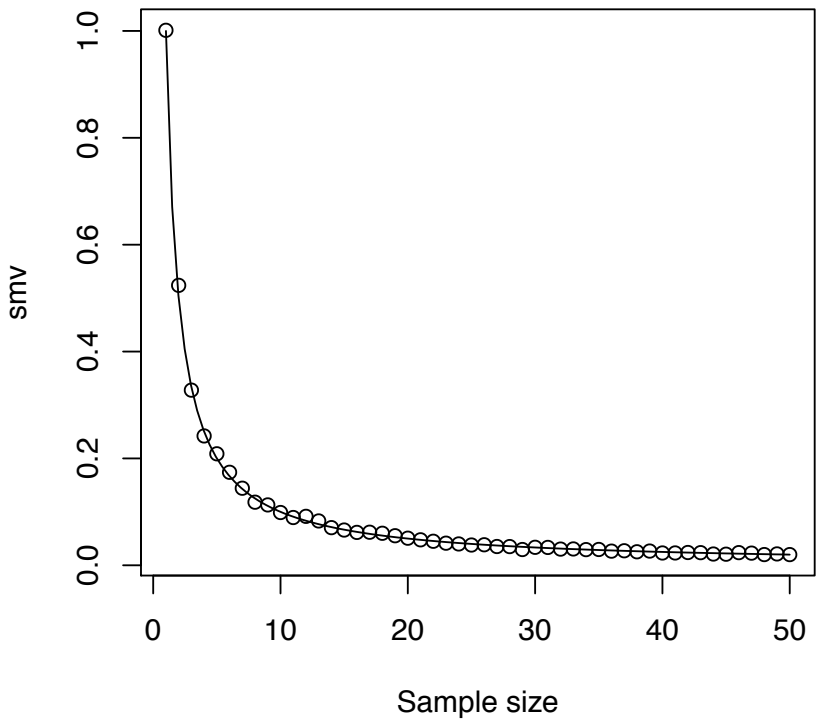
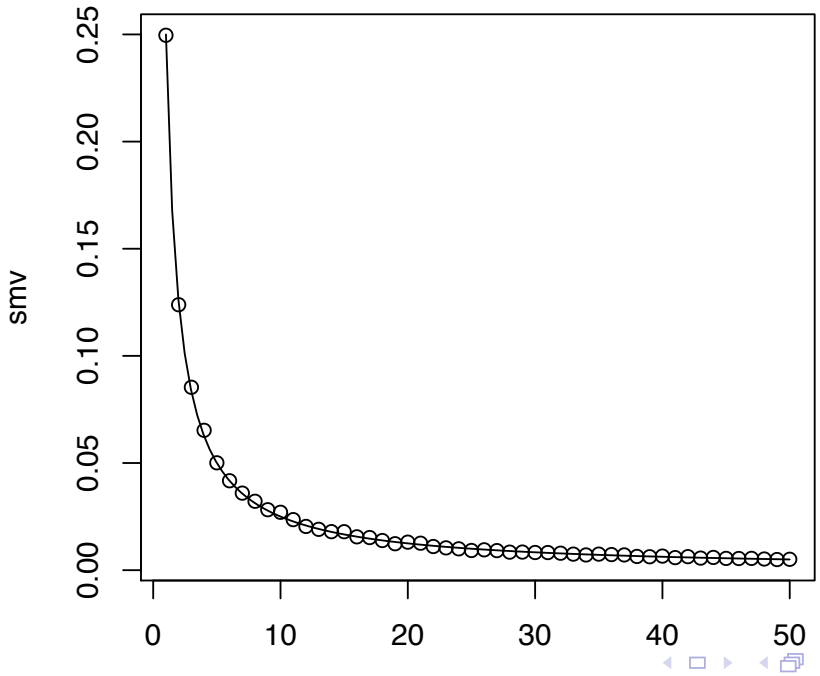


Illustration with binomial distribution

```
smv=sapply(1:50,
function(x) var(replicate(1000, mean(rbinom(x,1,0.5)))))
plot(1:50, smv, type="p", xlab="Sample size")
curve(0.25/x,1,50,add=T)
```



Outline

Motivation

- In R objects are defined by their class
- Several function adapt their "action" on the class of their arguments Example: **plot()**, **summary()**
- You can define your own object class, as well as the specific action generic functions should have on them.

Example: simple linear regression (again)

Example provided by Yves Croissant.

- First we specify that the output of function **slm()** is of class slm

```
slm=function(y,x){
b1=cov(x,y)/(var(x))
b0=mean(y)-b1*mean(x)
haty=b0+b1*x
e=y-haty
r2=var(haty)/var(y)
result=list(data=data.frame(y,x),coefficient=c(b0,b1),residuals=e,rsqua
class(result)="slm"
return(result) }
```

Example: simple linear regression (again)

- Then we can specify how generic function should treat slm objects

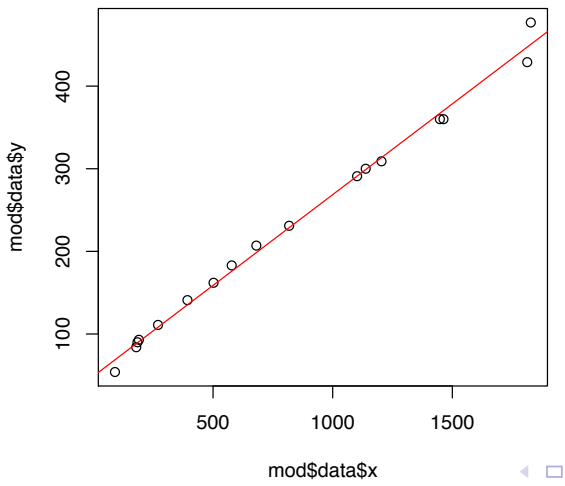
```
print.slm=function(mod, digits=3){
a=mod$coefficient[1]
b=mod$coefficient[2]
print(paste("The intercept is: ",round(a,digits)))
print(paste("The   slope   is: ",round(b,digits)))
}
###############
plot.slm=function(mod,...){
a=mod$coefficient[1]
b=mod$coefficient[2]
plot(mod$data$y~mod$data$x,...)
abline(a,b, col="red")
}
```

Result

```
setwd("~/Dropbox/Rprogramming/data")
data=read.table("airfares.txt", header=TRUE)
mod=with(data,slm(Fare,Distance))
print(mod)
```

```
[1] "The intercept is:  48.972"
[1] "The   slope   is:  0.22"
```

```
plot(mod)
```



The end

Thank you.