Introduction to R programming

Iván Andrés Trujilllo Abella

ai-page.readthedocs.ios

ivantrujillo1229@gmail.com

History of R

R is a dialect (an implemntation) of S language.

...

Created in 1991 in new zealand.

fragile

```
# The print function
print('hello world')
```

Assigment operation,

```
price <- 30 # assigment operator
quantity <- 5
income <- price * quantity
```

vectorized

R objects as used before are really vectors.

```r
output_msg <- "profit"
```

...

The object is character vector in R, the first is profit

[1]   10 a vector whose first element is the number ten
We'll create a sequence with colon(:) operator.

```
ito <- 1:10
```

R objects

atomic classes of objects:

- character
- integer
- numeric (float)
- complex
- logical (booleans)

keep information

vector - list

vector

each one of its elements are of the same class

list

can keep information of all kind

```
vector() # i don't have idea then;
help("vector")
```

Numbers

in R are generally treated as numeric objects (double precision real numbers).

```
123L # explitcit integer
```

Special

- inf (infinity)
- NaN (Not A Number)

Attributes

names, dimensions, class, lenght and another.

```
help("attributes")
attributes(df) # rows and columns
```

Vector - list

the $\mathrm{c}()$ (concatenate) function crate **vectors** of objects.

```
df <- c(0.9, 0.87, 0.95) # numeric object
booleans <- c(TRUE, FALSE, T, F) # logical
```

coercion

if you mix different kind of objects in a vector then occur coercion casting all the object to the same class, for instance c('1', 2) caste 2 to '2'.

Type

In programming languages define the type of data is very important thus:

```
bool <- c(True, F)
class(bool)
# explicit conversion
as.numerical(bool)
as.character(bool)
binaries <- c(1,0,1)
as.logical(binaries)
vowels <- c('a','e','i')
as.logical(vowels)
```

```r
data <- list('True', FALSE, 0 , 1.0)
```

Then the first element is a vector of characters, then second is a vector (logical) with the element FALSE, etc...

Matrices

```r
mat <- matrix(nrow=3, col= 2)
attributes(mat)
dim(mat)
```

```r
matrix(1:4,nrow=2,ncol=2) # column-wise (left-upper
    )
data <- 1:4
dim(data) <- c(2,2)
print(data)
```

cbind and rbind

```
x <- 5:10
y <- 50:55
cbind(x,y)
rbind(x,y)
```

Factors

Useful in categorical data, ordinal or cardinal, think in the problem itself that carry out pass categorical to numbers low=1, medium =2, high=3.

```
var <- factor(c('high', 'medium','low', 'low'))
table(var)
unclass(var) # how the factors are represented by R
```

Baseline factor

```
var <- factor(c('cs','math','cs' ,'economics',
'math','cs'),
leves=c('math', 'cs', 'economics'))
```

The first level is used as baseline level in some statistical procedures.

Missing values

NA and NaN

- NA (Not Avaliable) is general, a dont have information about missingness is a class!.
- NaN (Not a Number) arise from undefined mathematical operations as $\frac{a}{0}$

```
is.nan()
is.na()
```

a NaN (related to numerical) is also a NA but not viceversa.

The following is very important handled missing values.

```
data <- c(1,2,4, NA, NaN)
is.nan(data)
is.na(data
data <- c('1', 2, NaN, NA)
is.nan(data)
is.na(data)
```

DataFrame

Tabular data, important points:

- unlike matrices can handled different types of data (like list)
- could be comverted to matrices as df.matrix()

Create a dataframe to test:

```
df <- data.frame(var1=c('red','red','blue', 'blue'),
var2=c(T, T, F, F), var3=c(1,1,0,0))
# Describe the data
nrow(df)
ncol(df)
```

Names

```r
booleans <- c(T,F,T,F)
#Give name to each element...
names(booleans) <- c("yes", 'not', 'si', 'no')
names(booleans)
```

name and indexing

```r
data <- list(age= 40, id = 0, ibm =23)
data$age

mat <- matrix(1:6, ncol =3, nrow=2)
dimnames(mat)<-list(c('A', 'B'),
c('col1', 'col2', 'col3'))
# try
rownames(mat) <- c("A", "B")
colnames(mat) <- c('col1', 'col2', 'col3')
# indexing
mat['A', 'col3']
```

read and write common

- read.csv()
- read.table()
- wite.table()
- save()

Remeber that is very important get information in documentation `help("read.table")` also `read.csv()` is identical to `read.table` but if separator by deafult is the comma(,) character. important also remark that R skip lines that begin with # character.

subsetting

- [hold the class
- [[dont hold the class (think in lists)
- $ extract elemens (named) in list or dataframe

```
data <- c('Blue', 'Green', 'Red')
data[1] # take the first element (numerical
    index)
data[1:2]# from first to second
data[c(T,F,T)] #logical index
data[data=='Green'] # two equals signs (=).
data[data=='Green' | data=='Blue']
data <- c(5,10,11,2, 7)
data[data>6]
```

Research

...

- Research about the function `paste`
- Research about the function `cat`

subsetting

list

```
data <- list(var1= 5:10, var2 = 15:20)
data[1] + 1  # error is a list!
data[[1]] + 1  # numeric vector
class(data$var1)
class(data['var1'])
class(data[['var1']])
```

is important take care, given that could be produce errors!! Single Square Bracket always return an object of the same class of the original object. Double Square bracket with name is equal to dollar sign.

extract multiple elements

Uses single bracket operator

```
data <- list(a=10:30)
data$b <- 50:80
data$c <- 100:130
data[c("a","b")] # try with numerical index
# dollar sign not allow computed index for instance
    ;
column <- 'b'
data$column # return NULL
data[[column]] #execute!
data$b # execute!
```

subsetting nested elements.

```
data <- list(col1=list(T,F,T),
col2=list('Red','Blue'))
data[[c(1,2)]]
data[[1]][[3]]
```

Subsetting matrix

M_{ij} = `M[i,j]` where i is rows and j columnns.

```
mat <- matrix(1:10, 2,5)
mat[1,4]
mat[1,]
mat[,4]
class(mat[1,3])
class(mat[1,3, drop=F])
# A rule is broken [] only return the same class
class(mat[1,])
class(mat[1,,drop=F])
```

Removing missing values

```r
bools <- c(T,F)
!bools
data <- c(1, 0 , 0.5, NaN, NA)
data[!is.na(data)]
x <- c(NA, 3, NaN, 5, 10)
y <- c(10, 11, 4, Nan, 4)
complete.cases(x,y)
# assign to both arrays to extract the complete
    data.
```

in DataFrame

```
data <- data.frame(
age = c(32, NaN, 31, NaN),
score = c(NaN, 70, 81, 90))
index <- complete.cases(data)
data[index,]
```

Vectorized operations

```
x <- 10:15
y <- 0:5
x<y
x+y
x*y
x/y
```

```r
mat <- matrix(1:9, 3,3)
mat2 <- matrix(5:13, 3, 3)
mat*mat2 # Element wise multplication
mat %*% mat2 #Matrix multiplication
```

Control structures

```
if (condition)
{
statement 1
} else{
statement 2 (when cond is FALSE)
}
```

Example

```r
price <- 30
quantity <- 50
income <- price * quantity
if (income>10){
  print("Excellent!")
} else{
  print(':(')
}
```

else if

```
if (condition){
 Statement 1
} else if (condition) {
Statement alternative
} else{
else statement
}
```

...

```r
price <- 10
quantity <- 1
income <- price * quantity
if (income>10){
  print("Excellent!")
} else if (income == 10){
  print('equlibrium')
} else {
  print(':(')
}
```

loop

for

```
for (num in 1:10){
  print(num)
}
```

Examples

```r
x <- c(10, 0, 5)
for (i in x){
  print(i)
}

for (i in seq_along(x) ){
  print(i)
}

for (i in 1:length(x)){
  print(x[i] + 1)
}
```

Nested loop

```
for (i in 1:2) {
for (j in 3:5){
  print(c(i,j))
}}
```

As excercise nested a matrix! `ncol` and `nrow`.

Example

...

See simulation mean

See simulation estimator counter

See simulation

while loop

```r
acum <- 1
while (acum < 5) {
print(acum)
acum <- acum + 1
}
```

OR and AND

- && logical AND operator
- & element wise logical operator used in vectors to test AND
- || logical OR operator
- | element wise logical operator used in vectors to test OR

```
T && F
T || F
x <- c(T, T, F, F)
y <- c(T, F, T, F)
x&y
x|y
```

```r
# Search a number in the vector!
array <- c(10,40, 50, 1)
flag = FALSE
to_search = 1
index <- 1
while (index <= length(array) && flag==FALSE){
  print(array[[index]])
  if (array[[index]] == to_search){
    flag <- TRUE
  }
  index <- index + 1
}
flag
```

The conditions are evaluated from left to right.

Repeat, Next, Break

This topics are to research...

- `repeat` is a infinite loop whose exit condition is `break` statement.
- `next` skip a loop iteration (make an example with NAN)

Functions

```
function_name <- function(arguments){
function statement
}
```

```
future_val <- function(initial_val, r, n){
initial_val*(1 + r)^n #you could uses return
}
```

Excercise

...

write a function that compute the number of periods (n)

Excercise

…

write a function that compute the number of periods (n)

```
periods <- function(lambda, r){
return(log(lambda)/log(r +1 ))
}
```

Pi simulation

(Click here)

Practical

...

See lab

notes

Default arguments works like in python the `return` statement nos is necessary R return the last statement in the function body. THe arguments work as in python by name or position.
always check the arguments

...

args(print)

Important **lazy evaluation** evaluate arguments if they are needed.

```r
# lazy evaluation
power <- function(a,b,c){
  a^(b)
}
power(10,2)
#  error  not is arised
```

Important things in functions

... argument (ellipsis)

is important be careful using ellipsis (a undefined number of parameters).

```
suma <- function(...) {
args <- list(...)
acum <- 0
for (arg in args){
acum <- acum + arg}
acum
}
suma(1,2,3,4,5,6,7,8,9,10)
print(10*11/2)
```

ellipsis working as wrapped!

```
mean <- function(..., message='the mean is'){
  print(message)
  suma(...) / length(list(...))
}
mean(1,3,4) # arguments appear after of ellipses
    then they must be named explicity.
```

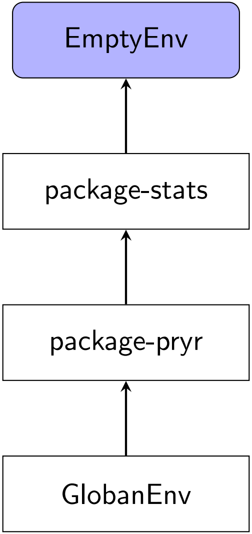
Scoping rules

The scope is the place where a variables is defined and its visibility.

- Lexical scoping
- Dynamic scoping

in R are exist *free variables*.

Figure: Enviroment tree



First the symbol: value is saerched in the global enviroment, and after in each on of the packages listed in the following command.

```
search()
```

This load all packages that are loaded currently in R. Global enviroment is always de fisrt and base the last, when you load a package the namespace occupy the second place and all others shifdown a position.

...

- R functions are stored and searched in the same way of other object, therefore are searched in the environment tree.

Annonymus functions

```
sapply(c(1,2,3,4), function(x) x^2)
```

$<< -$ operator

assign the variable to the global environment (not local where was defined).

```
local_var <- function(a){
    result <- a^2
    result
}
local_var(10)
result # object 'result' not found!

global_var <- function(a){
result <<- a^2
result
}
global_var(10)
result
```

Closures

Analyze

```
counter <- function(){
counter <- 0
function(){
counter <<- counter + 1
counter
}
}
acum <- counter()
acum()
acum()
acum()
```

Closures

...

- A function defined inside of another capture information from the outer function (its environment)
- The inner function can access and modify the variable count even though the function has already finished execution.

closures

```r
experiment <- function(sample_size){
mean <- 0
function(){
mean <<- mean(rnorm(sample_size))
mean
}
}
low_size <- experiment(10)
medium_size <- experiment(35)
for (i in 1:5){
print(c(low_size(), medium_size()))
}
```

Extend the function! to confidence intervals.

Closure

```
pi_estimator <- function(trials){
result <- function(){
acum <- 0
for (i in 1:trials){
if (sum(runif(2,-1,1)**2) <= 1){
acum <- acum + 1
}}
acum/trials*4
}}
small_trials <- pi_estimator(1000)
big_trials <- pi_estimator(1000000)
print(c(small_trials(), big_trials()))
```

Closures

Given lexical scoping `low_size` and `medium_size` can reference its environment definition!! and therefore the sample size defined 10 and 35 respectively.

```
ls(enviroment(low_size))
get('trials', environment(small_trials)) # knowing
    its enviroment!
```

Exercise

```
var <- 1
call <- function() var
change <- function() {
    var <- var +  100
    #var <<- var +  100
  call()
}
change() # the value is?
```

Important point!

define and after call a function in global environment resemble to dynamic scoping given that calling and defining environment are the same (add an example).
Analyze

```
y <- 3
f <- function(x){
y <- 2
y^2 + eval(expression(g(x)),envir = parent.frame()) # comment this line
#y^2 + g(x)  # uncomment this line
}
g <- function(x){
x^y
}
f(2)
```

Dates and time

...

- Dates are stores as the number of days since 01-01-1970
- Times as the number of seconds from 01-01-1970

```
date_ <- as.Date('1971-01-01')
unclass(date_)
```

Times

represented by POSIXct or POSIXlt

```
now <- as.POSIXlt(Sys.Date())
names(unclass(now))
now$mday
Sys.Date() # is in POSIXct format
# The number of seconds since 01-01-1970.
```

strptime function

Data is written in a different format

```
date_str <- c("December 29, 2024 9:10")
date_ <- strptime(date_str, "%B %d, %Y %H:%M")
date_
start <- as.Date("2021-01-03")
finish <- as.Date("2022-02-01")
finish - start # take care with the compatibility
```

Research time

...

- apply
- sapply
- lapply
- tapply
- mapply

Excersice Write standard deviation function

Excersice Write standard deviation function

```r
std_ <- function(..., ddof=0){
args <- as.numeric(list(...))
acum <- 0
mean_ <- mean(args)
for (arg in args){
  acum <- acum + (arg - mean_)**2
}
(acum / (length(args) - ddof))^0.5
}
print(std_(10,3,1,4, ddof=1))
print(sd(c(10,3,1,4)))
```

lapply

As an argument, receive a list (or cast it to one) and a function that receive as input each element of the list.

```
lapply(1:4, function(x){x^2})
```

```
lapply # ellipses represent arguments in the
    function.
data <- list(list(10,3,1),
list(10,3,1,4),
list(10,3))
lapply(data, function(x){sd(unlist(x))})
lapply(data, function(x){do.call(std_, x)}) #unpack
lapply(data, function(x) { do.call(std_,
c(x, ddof = 1)) }
```

List in R are "similar" to dictionary

```
lapply(list(col1=1:5, col2=10:15),
mean,trim=0, na.rm=T) # note that are passed  mean
    function arguments!
```

sapply

simplify the result of `lapply`

```
lapply(1:4, function(x) x**2)
sapply(1:4, function(x) x**2) # a vector is
    returned!
```

apply

`apply` usually used to apply a function over a row or column of a matrix. The first argument is data, the second is the axis(1 or 2) and the third the function.

```
mat <- matrix(c(0, 1, 10,
                1, 0, 3,
                5, 0, 0),
              nrow = 3, byrow = TRUE)
apply(mat, 1, sum) # sum all values in each row
apply(mat, 2, sum) # sum all values in each column
```

Research

...

- `rowSums`
- `colMeans`
- ...

how works

...

- `quantile`

...

- mapply
- tapply

split

Split objects into groups determined by factors.

```
data <- c(rnorm(5, 1), rnorm(5, 45), rnorm(5, 1000)
    )
f <- gl(3, 5) # three groups with five observations
     each one
split(data, f) # three groups
lapply(split(data,f), mean)
```

apply - split

```
df <- data.frame(income=c(10, 14, 15, 23.2, 21.3),
exp = c(6,8,3, 2, 4),
group=c('math','math','math','cs','cs'))
split(df, df$group)
lapply(split(df, df$group), function(x) colMeans(x
    [, c('income', 'exp')]))
```

more than one factor

```r
df <- data.frame(income=c(10, 14, 15, 23.2, 21.3, 21.4),
exp = c(6,8,3, 2, 4,5),
group=c('math','math','math','cs','cs', 'cs'),
sex = c('female', 'male', 'male', 'female', 'female', 'male'))

lapply(split(df, interaction(df$group, df$sex)), function(x) colMeans(x[, c('income', 'exp')]))
lapply(split(df, list(df$group, df$sex)), function(x) colMeans(x[, c('income', 'exp')])) #split allow us uses list
```

Debug

research about

- `traceback`
- `debug`
- `browser`
- `trace`
- `recover`

str

`str()` help us to visualize the data!

```
data <- list(1,2,3,4,5,6,7,8)
str(data)
str(df)
```

Simulation

```
set.seed(1) # uses seeds!
par(mar = c(4, 4, 1, 1))
domain <- seq(-3, 3 , length.out = 100)
range <- dnorm(domain)
plot(domain, range, type='l')
cumulative_range <- pnorm(domain)
plot(domain, cumulative_range, type='l')
```

Research statistica distributions

...

`help`("distribution")

- rpois
- dt
- dnbinom
- ...

prefixes

- d(ensity)
- r(andom number generator)
- p(robability - cumulative distribution)
- q(uantile function)

linear reg

```r
linear <- function(b0, b1, x, u){
b0 + b1*x + u
}
sample_size <- 500
b0 = 1
b1 = 3.4
x <- rnorm(sample_size, 10, 3)
u <- rnorm(sample_size, 0, 1)
y <- linear(b0, b1, x, u)
lm(y ~ x )
```

Sample

```
data <- c(100, 20, 5, 71, 89, 32, 121)
sample(data,2)
sample(data, 2, replace=T)
```