Laborarator 4 – Probabilități și Statistică Matematică

ELEMENTE DE PROGRAMARE IN R

4.1 Funcții

O *funcție* este un obiect în R care primește câteva obiecte de intrare (care se numesc *argumentele funcției*) și întoarce un obiect de ieșire. Structura unei funcții va avea următoarele patru părți:

- *Nume*: Care este numele funcției? Aveți grijă să nu folosiți nume ale funcțiilor deja existente în R!
- *Argumente*: Care sunt datele de intrare pentru funcție? Puteți specifica oricâte date de intrare doriți!
- *Corp sau acțiune*: Ce vreți să facă această funcție? Să traseze un grafic? Să calculeze o statistică?
- *Rezultat*: Ce vreți să vă întoarcă funcția? Un scalar? Un vector? Un data.frame?

```
# Structura de baza a unei functii
NUME <- function(ARGUMENTE) {

  ACTIUNI

  return(REZULTAT)

}
```

Funcțiile în R sunt *obiecte de primă clasă* (first class objects), ceea ce înseamnă că ele pot fi tratate ca orice alt obiect din R. Este important de reținut că, în R,

- funcțiile pot fi date ca argumente pentru alte funcții (de exemplu familia de funcții `apply()`)

- funcțiile pot fi imbricate (nested), cu alte cuvinte puteți crea funcții în interiorul altor funcții

Mai jos avem un exemplu de funcție care nu are niciun argument și nu întoarce nicio valoare:

```
f <- function() {
        ## Aceasta este o functie goala
}
## Functiile au clasa lor speciala
class(f)
[1] "function"

f()
NULL
```

Următoarea funcție întoarce numărul de caractere al textului dat ca argument:

```
f <- function(mesaj){
  chars = nchar(mesaj)
  chars
}

mes = f("curs de statistica si probabilitati")
mes
[1] 35
```

În fucția de mai sus nu am indicat nimic special pentru ca funcția să ne întoarcă numărul de caractere. În R, rezulatul unei funcții este întotdeauna ultima expresie evaluată. De asemenea există funcția `return()` care poate fi folosită pentru a întoarce o valoare explicită, dar de multe ori această funcție este omisă.

Dacă utilizatorul nu specifică valoarea argumentului `mesaj` în funcția de mai sus atunci R-ul întoarce o eroare:

```
f()
Error in nchar(mesaj): argument "mesaj" is missing, with no defa
ult
```

Acest comportament al funcției poate fi modificat prin definirea unei valori implicite (de default). Orice argument al funcției poate avea o valoare de default.

```
f <- function(mesaj = "Valoare de default"){
  chars = nchar(mesaj)
  chars
}

# Folosim valoarea implicita
f()
[1] 18
# Folosim o valoare specificata
f("curs de statistica si probabilitati")
[1] 35
```

- Să presupunem că Jack Sparrow este convins că poate prezice cât aur va găsi pe o insulă folosind următoarea ecuație: $ab-324c+\log(a)$, unde a este aria insulei (în m^2), b este numărul de copaci de pe insulă iar c reprezintă cât de beat este pe o scală de la 1 la 10. Creați o funcție numită `Jacks.Money` care primește ca argumente a, b și c și întoarce valoare prezisă.

Un exemplu ar fi

```
Jacks.Money(a = 1000, b = 30, c = 7)
[1] 27738.91
```

Argumentele funcțiilor în R pot fi potrivite după poziția lor sau după numele lor. Potrivirea după poziție înseamnă că R atribuie prima valoare primului argument, a doua

valoare celui de-al doilea argument, etc. De exemplu atunci când folosim funcție `rnorm()`,

```
str(rnorm)
function (n, mean = 0, sd = 1)
set.seed(1234) # pentru repetabilitate
mydata <- rnorm(10, 3, 1)
mydata
 [1] 1.7929343 3.2774292 4.0844412 0.6543023 3.4291247 3.5060559
2.4252600
 [8] 2.4533681 2.4355480 2.1099622
```

valoarea 10 este atribuită argumentului `n`, valoarea 3 argumentului `mean` iar valoarea 1 argumentului `sd`, toate prin potrivire după poziție.

Atunci când specificăm argumentele funcției după nume, ordinea acestora nu contează. De exemplu

```
set.seed(1234)
rnorm(mean = 3, n = 10, sd = 1)
 [1] 1.7929343 3.2774292 4.0844412 0.6543023 3.4291247 3.5060559
2.4252600
 [8] 2.4533681 2.4355480 2.1099622
```

întoarce același rezultat cu cel obținut mai sus.

De cele mai multe ori, argumentele cu nume sunt folositoare atunci când funcția are un șir lung de argumente și ne dorim să folosim valorile implicite pentru majoritatea dintre ele. De asemenea aceste argumente pot fi folositoare și atunci când știm numele argumentului dar nu și poziția în lista de argumente. Un exemplu de astfel de funcție este funcția `plot()`, care are multe argumente folosite în special pentru customizare:

```
args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
```

```
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes, panel.firs
t = NULL,
    panel.last = NULL, asp = NA, ...)
NULL
```

În R există un argument special notat `...`, care indică un număr arbitrar de argumente care sunt atribuite altor funcții din corpul funcției. Acest argument este folosit în special atunci când vrem să extindem o altă funcție și nu vrem să copiem întreaga listă de argumente a acesteia. De exemplu, putem crea o funcție de plotare în care specificăm tipul în prealabil

```
myplot <- function(x, y, type = "l", ...) {
        plot(x, y, type = type, ...)         ## Atribuie '...' f
unctiei 'plot'
}
```

Argumentul `...` poate fi folosit (și este necesar) și atunci când numărul de argumente pe care îl ia funcția nu este cunoscut în prealabil. De exemplu să considerăm funcțiile `paste()` și `cat()`

```
args(paste)
function (..., sep = " ", collapse = NULL)
NULL
args(cat)
function (..., file = "", sep = " ", fill = FALSE, labels = NULL
,
    append = FALSE)
NULL
```

Deoarece ambele funcții printează text în consolă combinând mai mulți vectori de caractere împreună, este imposibil ca acestea să cunoască în prealabil câți vectori de

caractere vor fi dați ca date de intrare de către utilizator, deci primul argument pentru fiecare funcție este `...`.

Este important de menționat că toate argumentele care apar după argumentul `...` trebuie explicitate după nume.

```
paste("Curs", "Probabilitati si Statistica", sep = ":")
[1] "Curs:Probabilitati si Statistica"
```

4.2 Structuri de control (`if-else`, `for`, etc.)

Structurile de control, în R, permit structurarea logică și controlul fluxului de execuție al unei serii de comenzi. Cele mai folosite structuri de control sunt:

- `if` și `else`: testează o condiție și acționează asupra ei
- `switch`: compară mai multe opțiuni și execută opțiunea pentru care avem potrivire
- `for`: execută o acțiune repetitivă de un număr fix de ori
- `while`: execută o acțiune repetitivă *cât timp* o condiție este adevărată
- `repeat`: execută o acțiune repetitivă de o infinitate de ori (trebuie să folosim `break` pentru a ieși din ea)
- `break`: întrerupe execuția unei acțiuni repetitive
- `next`: sare peste un pas în executarea unei acțiuni repetitive

4.2.1 `if-else`

Structura `if-else` este una dintre cele mai folosite structuri de control în R permițând testarea unei condiții și acționând în funcție de valoarea de adevăr a acesteia.

Forma de bază a acestei structuri este

```
if(<conditie>) {
        ## executa instructiuni atunci cand conditia este adevar
ata
}
else {
        ## executa instructiuni atunci cand conditia este falsa
}
```

dar putem să avem și o serie de teste, de tipul

```
if(<conditie1>) {
        ## executa instructiuni
} else if(<conditie2>)  {
        ## executa instructiuni
} else {
        ## executa instructiuni
}
```

Avem următorul exemplu

```
## Generam un numar uniform in [0,1]
x <- runif(1, 0, 10)

if(x > 3) {
        y <- 10
} else {
        y <- 0
}
```

4.2.2 Comanda `switch`

Comanda `switch` este folosită cu precădere atunci când avem mai multe alternative dintre care vrem să alegem. Structura generală a aceste comenzi este

```
switch (Expresie, "Optiune 1", "Optiune 2", "Optiune 3", ....., 
"Optiune N")
```

sau într-o manieră extinsă

```
switch (Expresie,
        "Optiune 1" = Executa aceste expresii atunci cand expres
ia se
                      potriveste cu Optiunea 1,
        "Optiune 2" = Executa aceste expresii atunci cand expres
ia se
                      potriveste cu Optiunea 2,
        "Optiune 3" = Atunci cand expresia se potriveste cu Opti
unea 3,
                      executa aceste comenzi,
        ....
        "Optiune N" = Atunci cand expresia se potriveste cu Opti
unea N,
                      executa aceste comenzi
)
```

Considerăm următorul exemplu

```
number1 <- 30
number2 <- 20
# operator <- readline(prompt="Insereaza OPERATORUL ARITMETIC: "
)


operator = "*"


switch(operator,
       "+" = print(paste("Suma celor doua numere este: ",
                         number1 + number2)),
       "-" = print(paste("Diferenta celor doua numere este: ",
```

```
                          number1 - number2)),
       "*" = print(paste("Inmultirea celor doua numere este: ",
                          number1 * number2)),
       "^" = print(paste("Ridicarea la putere a celor doua numer
e este: ",
                          number1 ^ number2)),
       "/" = print(paste("Impartirea celor doua numere este: ",
                          number1 / number2)),
       "%/%" = print(paste("Catul impartirii celor doua numere e
ste: ",
                           number1 %/% number2)),
       "%%" = print(paste("Restul impartirii celor doua numere e
ste: ",
                          number1 %% number2))
)
[1] "Inmultirea celor doua numere este:  600"
```

4.2.3 Bucle `for`

În R, buclele `for` iau o variabilă care se iterează și îi atribuie valori succesive dintr-un șir sau un vector. Buclele `for` au următoarea structură de bază

```
for (<i> in <vector>) {
        ## executa instructiuni
}
```

de exemplu

```
for(i in 1:10) {
        print(i)
}
[1] 1
[1] 2
```

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

Următoarele trei bucle prezintă același comportament

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
        ## Afiseaza fiecare elemnt din 'x'
        print(x[i])
}
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Funcția `seq_along()` este des întâlnită atunci când folosim bucle `for` deoarece crează un șir întreg folosind lungimea obiectului (în acest caz al lui `x`)

```
## Genereaza un sir folosind lungimea lui 'x'
for(i in seq_along(x)) {
        print(x[i])
}
[1] "a"
[1] "b"
[1] "c"
```

```
[1] "d"
```

De asemeanea putem folosi chiar pe `x` ca vector de indexare

```
for(letter in x) {
        print(letter)
}
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Atunci când folosim comenzile din buclele `for` pe o singură linie nu este necesară folosirea parantezelor `{}`

```
for(i in 1:4) print(x[i])
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

Putem folosi buclele `for` și imbricat (nested)

```
x <- matrix(1:6, 2, 3)

for(i in seq_len(nrow(x))) {
        for(j in seq_len(ncol(x))) {
                print(x[i, j])
        }
}
```

- Construiți următoarele matrice de dimensiune 10×10: $M_{i,j}=\frac{1}{\sqrt{|i-j|+1}}$ și

$N_{i,j} = \frac{i}{j^2}$. Puteți construi matricea M și matricea N fără a folosi bucle `for`?

(*Hint:* ce face comanda `outer`?)

4.2.4 Bucle de tip `while`

Acțiunile repetitive de tip `while` încep prin testarea unei condiții și în cazul în care aceasta este adevărată atunci se execută corpul comenzii. Odată ce corpul buclei este executat, condiția este testată din nou până când devine falsă (se poate ca bucla `while` să rezulte într-o repetiție infinită !). Structura generală a acestei bucle este

```
while(<conditie>) {
        ## executa instructiuni
}
```

Considerăm următorul exemplu

```
count <- 0
while(count < 4) {
        print(count)
        count <- count + 1
}
[1] 0
[1] 1
[1] 2
[1] 3
```

Uneori putem testa mai multe condiții (acestea sunt întotdeauna evaluate de la stânga la dreapta)

```
z <- 5
set.seed(123)

while(z >= 3 && z <= 10) {
```

```
        coin <- rbinom(1, 1, 0.5) # arunc cu banul

        if(coin == 1) {  ## random walk
                z <- z + 1
        } else {
                z <- z - 1
        }
}
print(z)
[1] 11
```

4.2.5 Bucle de tip `repeat`

Acest tip de acțiuni repetitive nu sunt foarte des întâlnite, cel puțin în statistică sau analiză de date. O situație în care ar putea să apară este atunci când avem un algoritm iterativ în care căutăm o soluție și nu vrem să oprim algoritmul până când soluția nu este suficient de bună.

```
x0 <- 1
tol <- 1e-8

repeat {
        x1 <- o_functie_definita()

        if(abs(x1 - x0) < tol) {  ## este suficient de buna solu
tia
                break
        } else {
                x0 <- x1
        }
}
```

4.2.6 Comezile `break` și `next`

Comanda `next` este folosită pentru a sării un pas într-o buclă

```
for(i in 1:10) {
        if(i <= 5) {
                ## sari peste primele 5 iteratii
                next
        }
  print(i)
      }
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Comanda `break` este folosită pentru a părăsi o buclă imediat

```
for(i in 1:10) {
      print(i)
      if(i > 5) {
              ## opreste dupa 5 iteratii
              break
      }     }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```