Funkce v R – užitečné vestavěné funkce a uživatelsky definované funkce v R

—

17VSADR – Skriptování a analýza dat v jazyce R

Lubomír Štěpánek[1, 2]

[1]Oddělení biomedicínské statistiky
Ústav biofyziky a informatiky
1. lékařská fakulta
Univerzita Karlova v Praze

ČVUT FBMI

[2]Katedra biomedicínské informatiky
Fakulta biomedicínského inženýrství
České vysoké učení technické v Praze

Obsah

Podmínka *když*

- též zvaná *if-statement*
- rozhodovací ovládací prvek založený na pravdivosti, či nepravdivosti nějakého výroku
- obecná syntaxe

```
1  if(výrok){
2    procedura v případě, že výrok je TRUE
3  }else{
4    procedura v případě, že výrok je FALSE
5  }
```

Podmínka *když*

- například

```
        if(x == 1){
          print("x je rovno 1")
        }

        # anebo
        if(x == 1){
          print("x je rovno 1")
        }else{
          print("x není rovno 1")
        }
```

for() cyklus

- ovládací struktura, smyčka pro opakování procedury stejného charakteru
- vhodná tehdy, **víme-li** dopředu počet opakování (iterací) dané procedury
- obecná syntaxe

```
1        for(indexový prostor){
2
3          procedura pro každý prvek indexového
4          prostoru
5
6        }
```

for() cyklus

- například

```
1   for(i in 1:5){
2
3     print(i)
4
5   }
6
7   # anebo
8   for(my_letter in letters){
9
10    print(
11      paste(my_letter, "je fajn", sep = " ")
12    )
13
14  }
```

while() cyklus

- ovládací struktura, smyčka pro opakování procedury stejného charakteru
- vhodná tehdy, **nevíme-li** dopředu počet opakování (iterací) dané procedury
- obecná syntaxe

```
1        index <- 1
2        while(výrok){
3
4          procedura pro každý index,
5          dokud je výrok TRUE
6
7          index <- index + 1
8
9        }
```

while() cyklus

- například

```
1  i <- 1
2  while(i <= 5){
3    print(i)
4    i <- i + 1
5  }
6
7  # anebo
8  my_letters <- letters
9  while(length(my_letters) > 0){
10
11   cat(
12     paste(my_letters[1], " je fajn\n", sep = "")
13   )
14   my_letters <- my_letters[-1]
15
16 }
```

repeat-until cyklus

- ovládací struktura, smyčka pro opakování procedury stejného charakteru
- vhodná tehdy, **nevíme-li** dopředu počet opakování (iterací) dané procedury
- prakticky ekvivalentní `while()` cyklu
- obecná syntaxe

```
1   index <- 1
2   while(TRUE){
3
4     if(zastavovací podmínka){break}
5
6     procedura pro každý index
7
8     index <- index + 1
9
10  }
```

repeat-until cyklus

- například

```
1  i <- 1
2  while(TRUE){
3    if(i == 5){break}
4    print(i)
5    i <- i + 1
6  }
7
8  # anebo
9  my_letters <- letters
10 while(TRUE){
11   if(length(my_letters) == 0){break}
12   print(
13     paste(my_letters[1], "je fajn", sep = " ")
14   )
15   my_letters <- my_letters[-1]
16 }
```

Varování

- textová hláška vrácená funkcí nebo procedurou
- nejde o maligní chybu

```
log(-5)  # NaN; In log(-5) : NaNs produced
```

- lze zavést i vlastní, například

```
logaritmuj <- function(x){
  # vrací přirozený logaritmus čísla "x"
  if(x <= 0){
    print(
      "x je nekladné, bude vráceno NaN"
    )
  }
  return(suppressWarnings(log(x)))
}

logaritmuj(-5)  # NaN; x je nekladné, bude vráceno NaN
```

Chyby

- ochranný mechanismus funkcí, který zabrání dalšímu provádění kódu

```
"1" + "1"  # Error: non-numeric argument to binary
           # operator
```

- lze zavést i vlastní, například

```
sectiCtverce <- function(a, b){
  # '''
  # vrací součet čtverců čísel "a" a "b"
  # '''
  if(!is.numeric(a)){stop("a musí být číslo!")}
  if(!is.numeric(b)){stop("b musí být číslo!")}
  return(a ^ 2 + b ^ 2)
}

sectiCtverce(1, 2)    # 5
sectiCtverce(1, "2")  # Error: b musí být číslo!
```

Úvod do funkcí v R

- funkce jsou důležité části kódu (a tedy programu)
- kromě funkcí má smysl odlišovat ještě procedury

```
1    is.function(print)      # TRUE
2    is.function("print")    # FALSE
```

Komponenty funkce

- funkce v R je složena ze tří částí
  - hlavička (*head* či *formals*)
    - obsahuje argumenty funkce
    - lze volat příkazem `formals()`
  - tělo (*body*)
    - obsahuje veškerý kód uvnitř funkce
    - lze volat příkazem `body()`
  - prostředí (*environment*)
    - mapuje lokalizaci všech proměnných ve funkci
    - lze volat příkazem `environment()`

```
1  f <- function(x){x^2}
2  f                    # function(x){x^2}
3
4  formals(f)           # $x
5  body(f)              # x^2
6  environment(f)       # <environment: R_GlobalEnv>
```

Primitivní funkce

- nemají tři uvedené komponenty
- ihned volají C-čkový kód pomocí `.Primitive()` a neobsahují R-kový kód, proto jsou exekučně rychlé
- např. `sum()`, `cumsum()`, `prod()` apod.

```
1  sum                      # function (..., na.rm = FALSE)
2                           # .Primitive("sum")
3
4  formals(sum)             # NULL
5  body(sum)                # NULL
6  environment(sum)         # NULL
```

Intermezzo

- s pomocí následujícího kódu můžeme získat všechny funkce balíčku base

```
my_objects <- mget(ls("package:base"),
                   inherits = TRUE)
my_functions <- Filter(is.function, my_objects)
```

Intermezzo

- s pomocí následujícího kódu můžeme získat všechny funkce balíčku base

```
my_objects <- mget(ls("package:base"),
                   inherits = TRUE)
my_functions <- Filter(is.function, my_objects)
```

- která z těchto funkcí má nejvíce argumentů?

Intermezzo

- s pomocí následujícího kódu můžeme získat všechny funkce balíčku base

```
my_objects <- mget(ls("package:base"),
                   inherits = TRUE)
my_functions <- Filter(is.function, my_objects)
```

- která z těchto funkcí má nejvíce argumentů?
- které z těchto funkcí nemají žádný argument?

Intermezzo

- s pomocí následujícího kódu můžeme získat všechny funkce balíčku base

```
my_objects <- mget(ls("package:base"),
                   inherits = TRUE)
my_functions <- Filter(is.function, my_objects)
```

- která z těchto funkcí má nejvíce argumentů?
- které z těchto funkcí nemají žádný argument?
- jak by bylo třeba změnit výše uvedený kód, abychom našli všechny primitivní funkce?

Scoping

- scopingem rozumíme prostor, ze kterého se v dané situaci a za daných okolností vybírají proměnné k výpočtu
- v R obecně *lexikální scoping*[1]
  - proměnné použité (ale nedefinované) ve funkcích se vybírají z mateřského prostředí (o jeden level výš)
- například

```
1  x <- 1
2  f <- function(){x}
3  g <- function(){
4       x <- 0
5       f()
6     }
7  g()   # 1
```

[1] kromě něj rozlišujeme ještě *dynamický scoping*, ten ale v R uplatňujeme jen u interaktivní volby proměnných pomocí funkce `attach()`

Scoping

- další příklad

```
1       x <- 1
2       f <- function(x){return(2 * x)}
3
4       f(x = 5)   # 10
5
6       x          # 1
```

- proměnné jsou tedy vnímány jako zástupné symboly, podobně jako ve středoškolské matematice
  - uživatelsky velmi přívětivé

Lexikální scoping

- objekty jsou ve funkcích hledány tak, jak byly funkce vnořeny do sebe při jejich vytváření, nikoliv jak jsou vnořeny při jejich volání
- v R je lexikální scoping zajištěn následovně
  - masking názvů
  - rozlišování proměnných a funkcí
  - fresh start
  - dynamický lookup

Masking názvů

- jsou-li všechny použité argumenty definovány ve funkci, je situace zřejmá

```
f <- function(){
    x <- 1; y <- 2
    c(x, y)
}
f()       # c(1, 2)
rm(f)
```

- pokud není některý z argumentů definován, R hledá o úroveň výš

```
x <- 1
g <- function() {
    y <- 2
    c(x, y)
}
g()       # c(1, 2)
rm(x, g)
```

Intermezzo

- čemu je rovno `h()`?

```
1  x <- 1
2  h <- function() {
3    y <- 2
4    i <- function() {
5      z <- 3
6      c(x, y, z)
7    }
8    i()
9  }
10 h()      # ???
```

Intermezzo

- čemu je rovno `h()`?

```
1  x <- 1
2  h <- function() {
3    y <- 2
4    i <- function() {
5      z <- 3
6      c(x, y, z)
7    }
8    i()
9  }
10 h()      # ???
```

```
1  h()      # c(1, 2, 3)
2  rm(x, h)
```

Rozlišování funkcí a proměnných

- dle kontextu (např. `f(3)`) dokáže R rozlišovat mezi funkcí a proměnnou stejného názvu

```
1  n <- function(x){x / 2}
2  o <- function(){
3       n <- 10
4       n(n)
5  }
6  o()    # 5
7  rm(n, o)
```

- přesto je lepší se takovému matoucímu kódu vyhnout

Fresh start

- při každém zavolání funkce je prostředí jejího těla (*body*) nově vytvořeno (tzv. *fresh start*)
- v následujícím příkladu bychom očekávali, že při prvním zavolání bude hodnota `j()` rovna 1, při druhém už 2, avšak díky fresh startu je pokaždé rovna 1

```
1   j <- function() {
2       if(!exists("a")){
3           a <- 1
4       }else{
5           a <- a + 1
6       }
7       a
8   }
9   j()     # pokaždé 1 díky fresh startu
10  rm(j)
```

Dynamický lookup

- hodnoty proměnných jsou hledány, když jsou funkce, které je používají, volány, nikoliv vytvářeny

```
1  f <- function(){x}
2  x <- 15
3  f()   # 15
4
5  x <- 20
6  f()   # 20
```

- takové chování je někdy nevýhodné, proto se někdy hodí znát vazby funkce na prostředí mimo prostředí jejího těla

```
1  library(codetools)
2  f <- function(){x + 1}
3  codetools::findGlobals(f)   # c("{", "+", "x")
```

Intermezzo

- co bude výsledkem následujícího kódu a proč?

```
1 ‘(‘ <- function(x){
2     if(is.numeric(x) & runif(1) < 0.1){
3         x + 1
4     }else{
5         x
6     }
7 }
8
9 replicate(50, (1 + 2))
```

Intermezzo

- co bude výsledkem následujícího kódu a proč?

```
1 ‘(‘ <- function(x){
2     if(is.numeric(x) & runif(1) < 0.1){
3         x + 1
4     }else{
5         x
6     }
7 }
8
9 replicate(50, (1 + 2))
```

- porovnejte nyní s

```
1 rm("(")
2
3 replicate(50, (1 + 2))
```

Lokální, globální proměnné

- koncept globálních proměnných se v R vůbec nedoporučuje používat, protože porušuje paradigma lexikálního scopingu
- přesto je možné zavést globální proměnnou přiřazením typu „<<-“

```
1        x <<- 1   # x je globální proměnnou
```

- ostatní proměnné jsou „lokální“

Pořadí argumentů v hlavičce funkce

- při volání funkce mohou být argumenty v její hlavičce specifikovány úplným názvem, částečným názvem, nebo pořadím (sestupně dle priority v tomto pořadí)

```
1  f <- function(ab, abc, b){
2      1 * ab + 2* abc + 3 * b
3  }
4  f(ab = 1, abc = 2, b = 3) # 14
5  f(abc = 2, ab = 1, b = 3) # 14
6  f(1, 2, 3)                # 14
7  f(2, 1, 3)                # 13
8  f(ab = 1, ab = 2, b = 3)
9     # formal argument "ab" matched
10    # by multiple actual arguments
11 f(a = 1, abc = 2, b = 3)   # 14
```

Volání funkce pro list argumentů

- máme-li list argumentů, pro který chceme volat danou funkci, lze využít příkaz `do.call()`

```
1  my_list <- list(1:10, trim = 0, na.rm = TRUE)
2
3  do.call(what = "mean", args = my_list) # 5.5
4
5  # odpovídá příkazu
6  # mean(1:10, trim = 0, na.rm = TRUE)
```

Iterování nad funkcemi

- předpokládejme, že chceme pro vektor hodnot `x` zjistit postupně průměr, minimum, maximum, medián, směrodatnou odchylku, varianci a vždy poslední cifru čísla

```
1  set.seed(1); x <- floor(runif(100) * 100)
2
3  for(my_function in c(
4      "mean",
5      "min",
6      "max",
7      "median",
8      "sd",
9      "var",
10     function(i) i %% 10
11 )){
12     print(do.call(my_function, list(x)))
13 }
```

Defaultní argument

- funkce v R mohou mít defaultní hodnoty

```
f <- function(a = 1, b = 2){c(a, b)}
f()   # c(1, 2)
```

- díky lazy evaluaci argumentů je lze definovat i závisle na sobě

```
g <- function(a = 1, b = a * 2){c(a, b)}
g()   # c(1, 2)
```

- dokonce lze definovat argumenty i pomocí proměnných, které vznikají až v těle funkce

```
h <- function(a = 1, b = d){
  d <- (a + 1) ^ 2; c(a, b)
}
h()   # c(1, 4)
```

Chybějící argument

- ve funkcích v R lze kontrolovat, zda argument chybí, pomocí příkazu `missing()`

```
1  i <- function(a, b){
2    c(missing(a), missing(b))
3  }
4  i()        # TRUE TRUE
5  i(a = 1)   # FALSE TRUE
6  i(b = 2)   # TRUE FALSE
7  i(1, 2)    # FALSE FALSE
```

Lazy evaluace

- ve funkcích v R je argument evaluován až tehdy, kdy nejdříve je to třeba

```
1  f <- function(x) {
2    10
3  }
4  f(stop("This is an error!")) # 10
```

- evaluaci lze forsírovat pomocí příkazu `force()`

```
1  f <- function(x) {
2    force(x)
3    10
4  }
5  f(stop("This is an error!")) # Error
```

Dot-dot-dot argument (...)

- umožňuje ve funkcích v R používat argumenty, které jsou unikátní jen pro některé z vnořených funkcí

```
1  f <- function(x, ...){
2    mean(x, ...)
3  }
4  f(c(0, 1, 2, 3, 3, 3, 3))
5    # 2.142857
6  f(c(0, 1, 2, 3, 3, 3, 3), trim = 1)
7    # 3
```

Vestavěné funkce

- většina funkcionality R je dána vestavěnými funkcemi
- ty jsou optimalizované, odladěné
- je-li to možné, je vhodné je preferovat před uživatelem definovanými funkcemi

Uživatelem definované funkce

- pro unikátnější operace je možné definovat vlastní funkce
- komponenty vlastní funkce jsou *hlavička s argumenty* a *tělo*
- obecná syntaxe

```
1   nazevFunkce <- function(
2       argument_1,
3       argument_2,
4       # ...
5   ){
6       # '''
7       # komentář
8       # '''
9
10      procedura s argumenty
11      return(výstup)
12  }
```

Uživatelem definované funkce

- například

```
1  sectiCtverce <- function(a, b){
2
3    # '''
4    # vrací součet čtverců čísel "a" a "b"
5    # '''
6
7    return(a ^ 2 + b ^ 2)
8
9  }
```

Intermezzo

- napište vlastní funkci, která rozhodne, zda je zadané přirozené číslo prvočíslem, nebo nikoliv

Rodina funkcí *apply()

- jde o funkce dobře optimalizované tak, že v rámci svého vnitřního kódu „co nejdříve“ volají C++ ekvivalenty R-kové funkce
- díky tomu jsou exekučně rychlé
- nejužitečnější je `apply()` a `lapply()`

```
1  # vrací průměry nad všemi sloupci "mtcars"
2  x <- apply(mtcars, 2, mean)
3
4  # méně šikovně to samé
5  x <- NULL
6  for(i in 1:dim(mtcars)[2]){
7    x <- c(x, mean(mtcars[, i]))
8  }
9  names(x) <- colnames(mtcars)
```

Funkce apply()

- vrací vektor výsledků funkce `FUN` nad maticí či datovou tabulkou `X`, kterou čte po řádcích (`MARGIN = 1`), nebo sloupcích (`MARGIN = 2`)
- syntaxe je `apply(X, MARGIN, FUN, ...)`

```
1  apply(mtcars, 2, mean)
2
3  my_start <- Sys.time()
4  x <- apply(mtcars, 2, mean)
5  my_stop <- Sys.time(); my_stop - my_start # 0.019s
6
7  my_start <- Sys.time()
8  x <- NULL
9  for(i in 1:dim(mtcars)[2]){
10   x <- c(x, mean(mtcars[, i]))
11   names(x)[length(x)] <- colnames(mtcars)[i]
12 }
13 my_stop <- Sys.time(); my_stop - my_start # 0.039s
```

Funkce lapply()

- vrací list výsledků funkce FUN nad vektore či listem X
- syntaxe je lapply(X, FUN, ...)
- **skvěle se hodí pro přepis for() cyklu do vektorizované podoby!**
- vhodná i pro adresaci v listu

```
1        set.seed(1)
2        my_long_list <- lapply(
3            sample(c(80:120), 100, TRUE),
4            function(x) sample(
5                c(50:150), x, replace = TRUE
6            )
7        )    # list vektorů náhodné délky
8             # generovaných z náhodných čísel
9
10       lapply(my_long_list, "[[", 14)
11            # z každého prvku listu (vektoru)
12            # vybírám jen jeho 14. prvek
```

Náhrada for cyklu funkcí lapply()

- obě procedury jsou ekvivalentní stran výstupu, `lapply()` je významně rychlejší

```
1  # for cyklus
2  x <- NULL
3  for(i in 1:N){
4    x <- c(x, FUN)
5  }
6
7  # lapply
8  x <- unlist(
9    lapply(
10     1:N,
11     FUN
12   )
13 )
```

Náhrada for cyklu funkcí lapply()

```
1  # for cyklus
2  my_start <- Sys.time()
3
4  for_x <- NULL
5  for(i in 1:100000){for_x <- c(for_x, i ^ 5)}
6
7  my_stop <- Sys.time(); my_stop - my_start # 18.45s
8
9  # lapply
10 my_start <- Sys.time()
11
12 lapply_x <- unlist(lapply(
13   1:100000, function(i) i ^ 5
14 ))
15
16 my_stop <- Sys.time(); my_stop - my_start # 0.10s
```

Literatura

Karel Zvára. *Základy statistiky v prostředí R*. Praha, Česká republika: Karolinum, 2013. ISBN: 978-80-246-2245-3.

Hadley Wickham. *Advanced R*. Boca Raton, FL: CRC Press, 2015. ISBN: 978-1466586963.

Děkuji za pozornost!

lubomir.stepanek@lf1.cuni.cz
lubomir.stepanek@fbmi.cvut.cz

▸ GitHub

github.com/LStepanek/17VSADR_Skriptovani_a_analyza_dat_v_jazyce_R