# Funkce

*Michal Kvasnička*

Funkce jsou základní stavební kámen v R: všechno, co se v R děje, je volání funkce. R ve svých balících nabízí velké množství funkcí. Často však potřebujete vytvořit funkce vlastní. V této lekci se podíváme na to k čemu funkce slouží, jak je vytvářet a jak je volat. Dostanete k dispozici také "slovník" vybraných funkcí, se kterými byste se měli seznámit.

## Funkce a jejich užití

Funkce je zapouzdřený kus kódu s jasným rozhraním (interface). Uživatel nemusí vědět nic o tom, co se děje uvnitř funkce – stačí mu vědět, jak funkci zavolat (jaké má jméno a jaké bere vstupní parametry) a jaké vrací hodnoty.

Existuje několik důvodů, proč vytváříme a používáme funkce:

- některý kód spouštíme znovu a znovu; pokaždé kód kopírovat je hloupé – kód je dlouhý a nepřehledný, při kopírování vznikne snadno chyba a kód se těžko mění (místo změny na jednom místě je třeba měnit totéž na mnoha místech)
- je dobré oddělit kód a jeho interface – pak můžeme vlastní kód snadno změnit; pokud se neměnil interface, je volání kódu stále stejné, takže nemusíme měnit nic jiného
- je snazší předat uživateli funkci než kód – uživatel (včetně našeho budoucího já) nemusí tušit, co je uvnitř, stačí mu, když ví, jak funkci spustit
- kód je modulárnější, úspornější a lépe se čte a udržuje

## Funkce v R

V R je funkce objekt jako jakýkoli jiný. To znamená, že

- vytvořenou funkci jde vložit do proměnné
- funkci jde předat jako parametr jiné funkci
- funkci můžeme vytvořit i uvnitř jiné funkce (nested function)
- jedna funkce může vracet jinou funkci jako svou hodnotu

Funkci tvoří tři části:

- interface funkce, tj. argumenty, které funkce bere; v R se to nazývá formals; funkce `formals()` vrátí seznam argumentů funkce
- tělo funkce, tj. kód funkce; funkce `body()` vrací kód uvnitř funkce
- prostředí (environment) funkce, které zahrnuje proměnné funkce; funkce `environment()` vrací prostředí funkce

Funkce v R mohou mít vedlejší účinky (side effects). Příkladem takové funkce je funkce `print()` – místo, aby vracela nějakou hodnotu, vypíše svůj argument nějakým způsobem do konzoly. Je jednodušší a bezpečnější psát čisté funkce bez vedlejších účinků.

## Tvorba funkce

Funkci vytvoříme tak, že výsledek vrácený funkcí `function()` uložíme do proměnné:

```
jmeno_funkce <- function(parametry funkce oddělené čárkami)
    výraz, který funkce vyhodnocuje
```

Pokud tělo funkce obsahuje víc než jeden výraz, je třeba tyto výrazy sbalit do bloku tím, že je uzavřeme do složených závorek:

```
jmeno_funkce <- function(parametry funkce oddělené čárkami) {
    výraz 1
    výraz 2
    ...
    výraz n
}
```

Funkce nemusí mít žádné parametry ani žádný kód. Pokud má funkce nějaký kód, pak vrací poslední vyhodnocený výraz jako svoji hodnotu:

```
nasobek <- function(x, y)
    x * y
class(nasobek)
```

```
## [1] "function"
```

```
nasobek(3, 4)
```

```
## [1] 12
```

```
nasobek(7, 8)
```

```
## [1] 56
```

Pokud je třeba vrátit hodnotu někde jinde než jako poslední výraz v těle funkce, stačí použít funkci `return()` – ta vrátí hodnotu funkce a zároveň ukončí její běh. (Pokud funkce vrací výsledek zabalený do funkce `invisible()`, funkce sice vrací hodnotu, ale při použití v konzole výsledek nevypíše.)

Funkce se vždy volá se závorkami, i kdyby neměla žádné parametry. Pokud zavoláme funkci bez závorek, vypíše se kód funkce (její tělo):

```
hello_world <- function()
    print("Ahoj, světe!")
hello_world
```

```
## function()
##     print("Ahoj, světe!")
```

```
hello_world()
```

```
## [1] "Ahoj, světe!"
```

Argumenty funkce mohou mít implicitní hodnoty – pokud není hodnota argumentu zadána, vezme se implicitní hodnota:

```
soucin2 <- function(x, y = 2)
    x * y
soucin2(3, 4)
```

```
## [1] 12
```

```
soucin2(3)  # y = 2, implicitni hodnota
```

```
## [1] 6
```

Parametry funkcí jsou vyhodnocovány líně (lazy evaluation). To znamená, že se jejich hodnota vyhodnotí až ve chvíli, kdy jsou opravdu použité. Pokud tedy není parametr ve funkci vůbec použit, R nevyhlásí chybu, když hodnotu parametru nezadáte.

```
f <- function(x, y)
    3 * x
f(2, 4)
```

```
## [1] 6
```

```
f(2)  # R nevyhlásí chybu, protože y není reálně použito
```

```
## [1] 6
```

Veškeré proměnné definované uvnitř funkce jsou lokální, tj. platí pouze uvnitř funkce a neovlivní nic mimo funkci. Při ukončení běhu funkce zaniknou (leda byste vytvořili uzávěru, viz dále). To platí i pro argumenty funkce – pokud se do nich uvnitř funkce pokusíte uložit nějakou hodnotu, R tiše vytvoří lokální proměnnou se stejným jménem, které zastíní vlastní parametr – jeho hodnota nebude ovlivněna.

```
a <- 3
b <- 7
f <- function(x, y) {
    a <- 5
    x <- 2 * x
    a + x + y
}
f(b, 3)  # vrací 5 + 2 * 7 + 3 = 22
```

```
## [1] 22
```

```
a  # hodnota a se mimo funkci nezměnila
```

```
## [1] 3
```

## Volání funkce

Funkce se volá svým jménem se závorkami, viz výše. Argumenty mohou být funkci předány třemi způsoby:

- jménem, např. `f(a = 1, b = 2)` – v tomto případě nezáleží na pořadí parametrů
- pozicí, např. ve funkci `f(a, b)` znamená volání `f(1, 2)`, že $a = 1$ a $b = 2$
- pokud má parametr implicitní hodnotu, je možné jej vynechat – R vezme místo parametru implicitní hodnotu

Při zadání parametru jménem R umožňuje jméno parametru zkrátit, pokud je zkratka jednoznačná. Např. ve funkci `f(number, notalk)` je možné první parametr zkrátit na `num` i `nu`, ovšem ne na `n`, protože `n` není jednoznačné – R by nevědělo, zda `n` znamená `number`, nebo `notalk`. Zkracování parametrů zjednodušuje interaktivní práci; při psaní skriptů se však výrazně nedoporučuje, protože autor funkce by později mohl přidat další parametr a zkrácené jméno už by nemuselo být jednoznačné.

Předávání parametrů těmito třemi typy jde libovolně míchat, tj. volat některé parametry pozicí, jiné jménem a další (s implicitní hodnotou) vynechat. V takovém případě postupuje R takto:

1. vezme parametry volané plným jménem (exact matching) a přiřadí jim hodnoty
2. vezme parametry volané zkráceným jménem (partial matching) a přiřadí jim hodnoty
3. vezme parametry pozičně

Pokud nechcete mít v kódu zmatek, doporučuji následující: první parametry funkce volat pozicí a jménem volat až parametry za nimi.

### Speciální argument ...

Kromě explicitně vyjmenovaných parametrů může mít funkce v R i speciální parametr `....` Tento parametr absorbuje libovolný počet parametrů na dané pozici; všechny parametry uvedené za ním musejí být volány plným jménem.

Parametr `...` se používá zejména ve dvou situacích: 1) když počet parametrů není dopředu znám a 2) když chceme některé parametry předat z naší funkce další funkci, kterou naše funkce volá.

Funkce `paste()` spojuje libovolný počet řetězců (bud spojuje vektory po prvcích, nebo kolabuje vektor do jednoho řetězce). Počet parametrů není znám dopředu, proto funkce paste používá parametr `...`:

```
paste("Ahoj,", "lidi.", "Jak se máte?")
```

```
## [1] "Ahoj, lidi. Jak se máte?"
```

```
paste
```

```
## function (..., sep = " ", collapse = NULL)
## .Internal(paste(list(...), sep, collapse))
## <bytecode: 0x21d20c8>
## <environment: namespace:base>
```

Funkce `print()` dokáže vypsat do konzoly obsah mnoha různých objektů – pro každý objekt volá speciální funkci přizpůsobenou tomuto objektu. Proto je třeba mít možnost funkci `print()` předat libovolné parametry, které funkce `print()` předá dál:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x34d90d0>
## <environment: namespace:base>
```

Pokud chcete napsat vlastní funkci s libovolným počtem parametrů, můžete použít následující trik: parametr `...` se vloží do seznamu, s jehož prvky se pak už pracuje obvyklým způsobem. Vytvoříme funkci, která vezme libovolný počet atomických vektorů a vrátí součet počtu prvků všech těchto vektorů:

```
count_all_members <- function(...) {
    param <- list(...)
    num <- sum(sapply(param, length))
    num
}
count_all_members(1)
```

```
## [1] 1
```

```
count_all_members(1, 1:10, 1:100, 1:1000)
```

```
## [1] 1111
```

(Co přesně dělá funkce `sapply()` vysvětlíme později.)

## Scoping rules a uzávěry

Když R hledá hodnotu nějaké proměnné, prochází série různých prostředí (environment). Pokud např. zadáte výpis proměnné `x` v konzoli, začíná R hledat `x` nejdříve v globální prostředí (tj. základním pracovním prostředí R). Když ji tam nenajde, pokračuje do mateřského prostředí (parental environment) globálního prostředí, což je typicky poslední načtený balík. Pokud ji nenajde ani tam, pokračuje do dalšího mateřského prostředí atd. Pokud ji nikde nenajde, vypíše R chybu. Seznam prostředí, která R prohledává, vrací funkce `search()`:

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "Autoloads"         "package:base"
```

Pokud nic nezměníte, vyhledává se nejdříve v globálním prostředí a nakonec v balíku **base**. Když načtete nový balík, vloží se na druhé místo a posune všechny balíky směrem dolů.

Situace ve funkcích je zajímavější. R používá tzv. lexical scoping, což znamená, že vyhledávání začíná v prostředí, ve kterém byla funkce definovaná. Pokud je funkce zadefinovaná v globálním prostředí, vše funguje normálně. Pokud však funkce definovaná uvnitř jiné funkce, pak vyhledávání začíná v prostředí této funkce, a pak v prostředí vnější funkce. To slouží k tvorbě tzv. function factories.

```
n <- 17
make.power <- function(n) {
    g <- function(x)
        x ^ n
    g
}
square <- make.power(2)
cube <- make.power(3)
square(2)
```

```
## [1] 4
```

```
cube(2)
```

```
## [1] 8
```

```
n
```

```
## [1] 17
```

Funkce `make.power()` vrací funkci jedné proměnné. Tyto funkce však obsahují kromě formálního parametru `x` také volný parametr `n`. Když R hledá jeho hodnotu, začne nejdříve v prostředí, ve kterém byly funkce `square()` a `cube()` definovány. Tato prostředí jsou prostředí funkce `make.power()` ve chvíli, kdy byla tato funkce spuštěna. Funkce `square()` byla definována v prostředí, ve kterém bylo `n = 2`, a toto prostředí si s sebou táhne. Stejně tak funkce `cube()` si s sebou táhne prostředí, ve kterém je `n = 3`. (Normálně volací prostředí funkce ve chvíli ukončení jejího běhu zaniká). Funkce `square()` a `cube()` jsou tzv. "uzávěry" (closures).

Existují situace, kdy je vytváření uzávěr užitečné. Jindy však uzávěry vzniknou, aniž byste si to výslovně přáli. To pak může být zdrojem překvapivých chyb.

## Řídící struktury a spol.

Zatím jsme předpokládali, že kód skriptu běží řádek po řádku. Někdy je však potřeba běh kódu různě modifikovat:

- některé řádky provést pouze, když je splněná určitá podmínka
- některé řádky provádět opakovaně
- zastavit běh skriptu s chybovým hlášením

### Větvení kódu

K provedení kódu, pouze pokud je splněná nějaká podmínka, slouží `if`:

```
x <- 1
if (x == 1)
    print("O.K.: x je jedna!")
```

```
## [1] "O.K.: x je jedna!"
```

Syntaxe: v závorce je logický výraz, na druhém řádku je kód, který se provede pouze v případě, že logický výraz má hodnotu `TRUE`.

Pokud se má provést více než jeden řádek kódu, je třeba jej seskupit pomocí složených závorek:

```
if (x == 1) {
    a <- 5
    print("O.K.: x je jedna!")
}
```

```
## [1] "O.K.: x je jedna!"
```

```
a
```

```
## [1] 5
```

Pokud se má nějaký kód provést při platnosti podmínky a jiný, pokud podmínka neplatí:

```
if (x == 1) {
    print("O.K.: x je jedna!")
} else {
    print("O.K.: x není jedna!")
}
```

```
## [1] "O.K.: x je jedna!"
```

Pozor: `else` musí být na stejném řádku, jako končící složená závorka nebo kód, který se provádí při splnění podmínky.

### Opakování

Pokud se má nějaký kus kódu opakovat $x$-krát, použijte `for`.

Pokud se má nějaký kus kódu opakovat, dokud je splněná nějaká podmínka, použijte `while` nebo `repeat`, viz dokumentace.

### Zastavení kódu a výstrahy

K zastavení běhu skriptu slouží funkce `stop()`: zastaví běh skriptu s chybou a vypíše svůj argument:

```
# kód se zastaví, pokud v není řetězec (zde se nezastaví)
v <- "ahoj"
if (!is.character(v))
    stop("v není řetězec!")
```

Zprávy je možné do konzoly vypsat pomocí funkce `message()`, varování pomocí funkce `warning()`:

```
if (!is.list(v))
    warning("Pozor: v není seznam!")
```

```
## Warning: Pozor: v není seznam!
```

Do konzoly je samozřejmě možné vypisovat i pomocí funkcí `print()`, `cat()` apod. Zprávy o běhu kódu však vypisujte raději pomocí `message()` a `warning()`: jsou barevně odlišené (aspoň v RStudiu) a je možné je snadno potlačit, pokud nejsou žádoucí, což v případě `print()` a spol. nejde.

## Seznam užitečných funkcí

K samostudiu s využitím dokumentace...

Výběr ze slovníku z kapitoly v knize Hadleyho Wickhama *Advanced R* dostupné na http://adv-r.had.co.nz/Vocabulary.html, více najdete v tam.

### Struktura datových objektů

```
str
```

### Výpis dat a zpráv

```
print, cat
head, tail
message, warning
format
sink, capture.output
```

### Srovnání a logické operace

```
&, |, !, xor
all.equal, identical
!=, ==, >, >=, <, <=
is.na, complete.cases
is.finite
```

### Základní matematika

```
*, +, -, /, ^, %%, %/%
abs, sign
acos, asin, atan, atan2
sin, cos, tan
ceiling, floor, round, trunc, signif
exp, log, log10, log2, sqrt
```

**Základní statistika**

```
max, min, prod, sum
cummax, cummin, cumprod, cumsum, diff
pmax, pmin
range
mean, median, cor, sd, var
sample
choose, factorial, combn
```

**Množinové funkce**

```
%in%
all, any
intersect, union, setdiff, setequal
which
```

**Vektory a matice**

```
c, matrix
length, dim, ncol, nrow
cbind, rbind
names, colnames, rownames
t
diag
as.matrix, data.matrix
rep, rep_len
seq, seq_len, seq_along
rev
```

**Test a konverze datového typu**

```
(is/as).(character/numeric/logical/...)
```

**Seznamy a datasety**

```
list, unlist
data.frame, as.data.frame
split
expand.grid
```

### Faktory

```
factor, levels, nlevels
reorder, relevel
cut, findInterval
interaction
```

### Pořadí a tabulace dat

```
duplicated, unique
merge
order, rank, quantile
sort
table, ftable
```

### Náhodné veličiny

```
(q, p, d, r) * (beta, binom, cauchy, chisq, exp, f, gamma, geom,
  hyper, lnorm, logis, multinom, nbinom, norm, pois, signrank, t,
  unif, weibull, wilcox, birthday, tukey)
```

Všechny tyto funkce existují vždy ve čtyřech verzích: začínající `d` vrací hustotu pravděpodobnosti, `p` vrací kumulativní pravděpodobnostní funkci, `q` vrací daný kvantil a `r` generuje náhodná čísla s daným rozdělením. Příklad pro normální rozdělení (`norm`):

```
x <- rnorm(10)  # 10 náhodných čísel, x[i] ~ IID N(0, 1)
x
```

```
##  [1] -0.15851089 -0.35521399  1.48251158  0.76427098  1.43516560
##  [6] -2.18134249  0.48558271 -0.08827887  0.78559658 -1.49509857
```

```
dnorm(x)  # hustota pravděpodobnosti x ~ IID N(0, 1)
```

```
##  [1] 0.39396178 0.37455111 0.13293981 0.29790114 0.14244651 0.03695457
##  [7] 0.35457556 0.39739079 0.29301850 0.13047177
```

```
y <- pnorm(x)  # kumulativní pravděpodosbnost x ~ IID N(0, 1)
y
```

```
##  [1] 0.43702712 0.36121464 0.93089789 0.77764711 0.92438004 0.01457905
##  [7] 0.68636847 0.46482751 0.78394807 0.06744436
```

```
qnorm(y)  # kvantil y ~ IID N(0, 1), totéž co x
```

```
##  [1] -0.15851089 -0.35521399  1.48251158  0.76427098  1.43516560
##  [6] -2.18134249  0.48558271 -0.08827887  0.78559658 -1.49509857
```

Parametrizace rozdělení viz dokumentace.

### Maticová algebra

```
crossprod, tcrossprod
eigen, qr, svd
%*%, %o%, outer
rcond
solve
```

#### Pracovní prostředí

```
ls, exists, rm
getwd, setwd
q
source
install.packages, library, require
```

#### Nápověda

```
help, ?
help.search
apropos
RSiteSearch
citation
demo
example
vignette
```

#### Čtení a zápis dat

```
data
count.fields
read.csv, write.csv
read.delim, write.delim
load, save
```

#### Soubory a adresáře

```
dir
basename, dirname, tools::file_ext
file.path
path.expand, normalizePath
file.choose
file.copy, file.create, file.remove, file.rename, dir.create
file.exists, file.info
tempdir, tempfile
download.file, library(downloader)
```

## Domácí úkol

Upravte soubor `hw04.R`. Vytvořte v něm funkci `xMax(v, n)`, která vrátí $n$-tý největší prvek vektoru `v`. Vektor `v` musí být zadán vždy. Pokud je `n` zadáno, vrátí `n`-tý největší prvek; pokud `n` není zadáno, vrátí druhý největší prvek.

Funkce skončí chybou, pokud

- `n` je zadáno, ale není to jedno celé číslo (pozor: nemusí být třída integer!, chyba `"n isn't one round number!"`),
- `n` je menší než jedna nebo větší než délka vektoru `v` (chyba `"n is out of bounds!"`)
- `v` není numerický atomický vektor (chyba `"v is not numeric atomic vector!"`)
- nejsou hodnoty vektoru `v` unikátní, tj. některá hodnota se ve vektoru vyskytuje vícekrát (chyba `"v doesn't have unique values!"`); hint: funkce `unique()`

Při ukončení chybou musí funkce vypsat předepsanou chybovou hlášku.