Tvorba vlastních funkcí

Funkce je pojmenovaná řada operací. Ve funkci je nadefinováno, co má udělat s objekty, na které ji použijeme. Abychom tuto definici mohli nějak zapsat, vytvoříme si imaginární jména objektů, které do funkce budou vstupovat a pomocí těchto imaginárních objektů zapíšeme, co s nimi má funkce udělat. Při použití dané funkce pak jen **R** sdělíme, co si má za imaginární objekty dosadit. Imaginární objekty jsou tedy vlastně **argumenty** dané funkce.

Příklad: vytvoříme funkci, která bude počítat aritmetický průměr hodnot nějakého vektoru. Funkce bude pracovat s jedním objektem, s vektorem hodnot. Zvolím si jméno imaginárního vektoru (jediného argumentu funkce), např. x, ať je to krátké. Průměr je součet hodnot vydělený jejich počtem. S použitím imaginárního jména vektoru x bychom operaci v **R** zapsali např. takto: `sum(x) / length(x)`. Teď už zbývá **R** jen sdělit, že danou operaci má uložit jako funkci imaginárního objektu (argumentu) x:

```
mymean<- function(x){
  sum(x) / length(x)
}
```

Teď můžeme funkci `mymean()` použít jako jakoukoliv jinou funkci. Má argument x, který představuje vektor hodnot, na kterých se má spočítat aritmetický průměr.

```
mymean(x= 1:10)

## [1] 5.5

mymean(c(2,4,1,0,0,3,4,8))

## [1] 2.75
```

Pokud naše funkce používá jiné již nadefinované funkce, je často vhodné ponechat prostor pro použití dalších, volitelných argumentů oněch funkcí. Například naše funkce neumí počítat aritmetický průměr, pokud se ve vektoru vyskytne NA hodnota. `mymean()` vrátí NA, protože funkce `sum()` neví, jakou hodnotu má sčítat:

```
mymean(c(1,5,NA,4))

## [1] NA
```

Bylo by proto rozumné ponechat možnost pro proužití argumentu `na.rm=`u funkce `sum()`. K tomu slouží 3 tečky `...`, které najdete v definici naprosté většiny funkcí. Tyto tři tečky umožňují volitelné použití argumentů funkcí použitých uvnitř dané funkce a značí "a další argumenty, které je možné vložit do použitých funkcí". `...` umístíme jednak do funkce `function()`, kde vyjmenováváme argumenty naší funkce, a pak jako další argument použitých funkcí tam, kde bychom mohli chtít použít nějaký další argument. Funkci teď samozřejmě musíme upravit tak, aby délku vektoru zjišťovala jen na hodnotách, které nejsou NA.

```
mymean<- function(x, ...){
sum(x, ...) / length(x[!is.na(x)])
}
mymean(c(1,5,NA,4), na.rm= T)

## [1] 3.333
```

Další možností by bylo použít argument `na.rm=` přímo v definici naší funkce. Pro demonstraci libovolnosti zvolených jmen ho pojmenuji `odstranNA`, ale pro zachování konzistence s ostatními funkcemi by bylo lepší ho pojmenovat stejně, `na.rm`.:

```
mymean<- function(x, odstranNA){
 sum(x, na.rm= odstranNA) / length(x[!is.na(x)])
}
mymean(c(1,5,NA,4), odstranNA= T)

## [1] 3.333
```

Jednotlivým argumentům můžeme při definici funkce přiřadit výchozí hodnoty. Například bychom mohli chtít, aby naše funkce implicitně odstraňovala NA hodnoty při počítání průměru. Pokud nastavíme výchozí hodnotu (např. `odstranNA= T`), nemusíme argument při použití funkce vůbec použít, v případě, že souhlasíme s danou přednastavenou hodnotou a použijeme jej jen pokud hodnotu chceme změnit (stejně jako je tomu u všech ostatních funkcí).

```
mymean<- function(x, odstranNA= T){
 sum(x, na.rm= odstranNA) / length(x[!is.na(x)])
}
mymean(c(1,5,NA,4))

## [1] 3.333

mymean(c(1,5,NA,4), odstranNA= F)

## [1] NA
```

Při programování se ještě může hodit možnost upravit chování funkce podle vložených argumentů. Například funkce, která počítá faktoriál, by měla vrátit produkt (násobek všech hodnot) sekvence celých čísel od 1 do x (hodnota, pro kterou faktoriál počítáme) v případě, že x je větší než 0. Pokud $x = 0$, má vrátit 1. Uděláme to tak, že formulujeme podmínku a definujeme operaci, kterou má funkce provést v případě, že je podmínka splněna. Syntaxe je následovná:

```
if(podminka){
operace
}
```

U našeho faktoriálu bychom mohli definovat tři operace: vrátit 1, pokud $x = 0$; vrátit produkt sekvence od 1 do x, pokud je $x > 0$, a vygenerovat chybovou hlášku, pokud x je záporné :

```
myfactorial<- function(x){
  if(x==0){
    return(1)
  }
  if(x>0){
    return(prod(seq(1:x)))
  }
  if(x<0){
    stop("zaporne 'x'")
  }
}
myfactorial(5)

## [1] 120

myfactorial(0)
```

```
## [1] 1

myfactorial(-5)

## Error: zaporne 'x'
```

Všimnite si ještě funkce `return()`, kterou explicitně **R** říkáme, co má vrátit. Funkce `return()` by měla obsahovat všechny výsledky, které chceme vrátit. Pokud bychom si přáli zobrazovat na obrazovku například mezivýsledky, použijeme funkci `print()`, pro upravená hlášení s textem pak `cat()`.

Možností při programování funkcí je samozřejmě mnohem více, pro běžnou tvorbu funkcí pro vlastní potřebu si však zhruba vystačíme s tímto. Pokud vás zajímá víc, mrkněte na `?Control`, `?stop`, `?print`.

Jak používat vlastní funkce

Nově vytvořenou funkci, aby ji bylo možné použít, je potřeba projet v **R**. Můžeme ji do **R** normálně poslat jako jakýkoliv jiný příkaz, ale není to moc šikovné, protože nám zbytečně zaplácne místo v konzoli. Pak také, aby naše práce zachycená ve scriptu aktuálního .r souboru byla reprodukovatelná, měl by celý kód té fukce být v tom scriptu obsažen. Ideální řešení nabízí funkce `source()`:

1. pro napsání funkce si otevřeme nové scriptové okno, ve kterém funkci napíšeme a dokumentujeme poznámkami za #,
2. funkci pak uložíme jako samostatný .r soubor,
3. pro použití oné funkce si nazdrojujeme (nasourcujeme) daný soubor. Pomocí funkce `source()` tak projedeme obsah celého souboru v **R** bez toho, aby se nám zaplácala konzole nebo aktuální .r script, přitom ale bude ve scriptu zaznamenáno, že jsme ten soubor nasourcovali.

Sourcování pak je analogické třeba načtení dat, jen použijeme funkci `source()` a specifikujeme cestu k sourcovanému souboru, například `source("D:/MyFun/myfactorial.r")`. V RStudiu můžeme mít otevřených víc scriptových oken zároveň. Můžeme funkci upravovat a opravovat, po té ji vždy uložíme a znovu nasourcujeme a vyzkoušíme v našem pracovním okně.