Higher order functions and libraries

Tamas Kadlecsik

March 11, 2016

> "To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs."
>
> — Bjarne Stroustrup

A magasabb rendű függvények olyan függvények, amelyek legyalább egyik paramétere függvényt vár és / vagy visszatérési értékük egy függvény. A második fajta, függvényt visszaadó függvény kicsit bonyolultabb, azt csak azoknak ajánlom, akik már legalább 2-3 éve programoznak. Azok pedig nyugodtan nézzenek utána más programnyelveknek, mint például: F#, Haskell, OCaml, erlang, Lisp, illetve olyan koncepcióknak, mint "partial application", "currying" és "function composition".

Mi most inkább maradjunk az elsőnél: olyan függvény, aminek legalább az egyik paramétere függvényt vár.

Írjunk először egyet saját magunk!

```
higher.order <- function(func, x){
  result <- func(x)
  return(result)
}

higher.order(sum, 1:10)
```

```
## [1] 55
```

```
sum(1:10)
```

```
## [1] 55
```

Hurrá! Már megint egy marha értelmes függvény. Nem a haszna a lényeg, hanem hogy kaptunk egy igen erőteljes és hasznos koncepciót! Szedjük kicsit szét ez a függvény rögvest.

Maga a függvény semmi mást nem csinál, mint vár egy egy változós – **unary** – függvényt, illetve egy objektumot, majd végrehajta a kapott onjektumon a kapott függvényt és visszadja az eredményt.

Esetünkben ez a sum függvén (függvényhívó `()` operátor nélkül!) illetve az `1:5` vektor.

Ha egy kicsit általánosítani akarjuk, várhatunk meghatározatlan számú argumentumot, és mindet átadhatjuk a kapott függvénynek:

```
higher.order <- function(func, ...){
  result <- func(...)
  return(result)
}

higher.order(sum, 1:10)
```

```
## [1] 55
```

```r
sum(1:10)
```

```
## [1] 55
```

```r
higher.order(paste, 'experiment', 1, '.csv', sep='')
```

```
## [1] "experiment1.csv"
```

```r
paste('experiment', 1, '.csv', sep='')
```

```
## [1] "experiment1.csv"
```

Az első argumentum egy bármilen függvény, a többi pedig a számára tartogatott argumentumok. Na de vajon mire lehet ez nekünk jó?

Nekünk nem feltétlenül kell magasabb rendű függvényeket írnunk, azonban az R-be igen sok ilyen függvény van beépítve, így nem árt, ha van elképzelésünk róla, hogy hogyan is működnek.

Definiáljunk egy saját függvény, és adjuk át azt a higher.order()-nek!

```r
test.function <- function(x, y, z){
  return((x+y)/z)
}

higher.order <- function(x, fun, ...){
  print(x)
  result <- fun(x, ...)
  return(result)
}

higher.order(1:10, test.function, 7, 8)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
##  [1] 1.000 1.125 1.250 1.375 1.500 1.625 1.750 1.875 2.000 2.125
```

Vagyis az 1:10 vektor minden eleméhez hozzáadott 7-et, majd elosztotta 8-cal.

$$\frac{1+7}{8} = 1 \frac{2+7}{8} = 1.125 \frac{3+7}{8} = 1.250...$$

és így tovább.

A `higher.order` megírásakor még nem tudhatjuk, mennyi paramétere lesz a kapott függvénynek, de biztosítanunk kell, hogy a kapott extra paraméterek átkerülnek átadódnak a függvénynek.

aggregate(formula, data, FUN, ..., subset, na.action = na.omit)

Elsőként nézzük meg az igen hasznos aggregate() függvényt! Ez egy data.frame-en belüli oszlop numerikus értékeit aggregálja egy adott függvényt felhasználva, egy csoportváltozó szerint. Huh, na nézzük

```
experiment <- read.csv("../../3_file_handling/homework_1/experiment.csv")
aggregated <- aggregate(weight ~ treatment, data=experiment, FUN=mean)
aggregated
```

```
##   treatment   weight
## 1   control 14.95815
## 2   treated 14.73232
```

Na akkor mégegyszer: Egy adott data.frame-en belül : experiment egy adott függvényt felhasználva : FUN=mean egy oszlop numerikus értékeit: weight egy csoport változó szerint: treatment aggregálja

Megint huh.

Több numerikus értéket is vizsgálhatunk, ha cbinddal összefüzzük őket:

```
experiment$life.span <- experiment$died - experiment$born
aggregated <- aggregate(cbind(weight, life.span) ~ treatment, data=experiment,  FUN=mean)
aggregated
```

```
##   treatment   weight life.span
## 1   control 14.95815  1252.172
## 2   treated 14.73232  1247.978
```

Több csoportváltozót is megadhatunk a + operátor segítségével:

```
experiment$life.span <- experiment$died - experiment$born
aggregated <- aggregate(cbind(weight, life.span) ~ treatment + diet, data=experiment,  FUN=mean)
aggregated
```

```
##   treatment    diet   weight life.span
## 1   control control 14.87960  1255.691
## 2   treated control 14.80942  1251.740
## 3   control treated 15.04717  1248.183
## 4   treated treated 14.63657  1243.306
```

Ha sok különböző numerikus változót szeretnénk vizsgálni érdemes először a `cbind()` hívás értékét változóba elrakni. A csoportváltozókkal ezt sajnos nem tehetjük meg.

A paraméterlistában FUN után álló ..., arra szolgál, hogy ha a mean-nél összetettebb függvényt szeretnénk használni, aminek esetleg még egyéb paraméterei is vannak, az aggregálandó objektumon kívül, akkor azokat ott megadhatjuk.

Apply-ok

`apply()`

Az apply-ok egy R objektumot, és egy függvényt várnak, majd a függvényt végrehajtják az objektum összes elemén.

A legelső apply függvény maga az apply(). Ez mátrixok sorain vagy oszlopain hajt végre műveleteket, de annyira nem izgalmas, és ki is lóg a sorból, úgyhogy ugorjunk a lényegre!

`lapply()`

Sokkal szimpatikusabb, ha kifejtjük az "l" betűt: list apply. Amúgy az összes többi apply neve előtt egy betű lesz, éppen ezért összefogaló néven `*apply`-nak is szokták hívni ezeket a függvényeket. A többi `*apply` függvény igazából ennek a burkolója.

```
test.list <- list(a = 1, b = 1:3, c = 10:100)

lengths <- lapply(test.list, length)
lengths
```

```
## $a
## [1] 1
##
## $b
## [1] 3
##
## $c
## [1] 91
```

```
sums <- lapply(test.list, sum)
sums
```

```
## $a
## [1] 1
##
## $b
## [1] 6
##
## $c
## [1] 5005
```

Ha ugyanezt ciklussal szeretnénk megoldani:

```
test.list <- list(a = 1, b = 1:3, c = 10:100)

lengths <- list()
for(element in test.list){
  lengths <- c(lengths, length(element))
}

lengths
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 3
##
## [[3]]
## [1] 91
```

```
sums <- list()
for(element in test.list){
  sums <- c(sums, sum(element))
}

sums
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 5005
```

Az előző két ciklust nyugodtan összevonthatnánk egyé is. Azonban a `lapply` hívás jóval áttekinthetőbb kódot biztosít, ráadásul védve van minden mellékhatástól.

A `lapply`-t bármilyen változón alkalmazhatjuk, ami listává alakítható, így vektoron, mátrixon, és data.frame-en is, azonban eredményül mindig listát fogunk kapni.

sapply()

A `*apply()` család másik gyakran alkalmazott tagja. Ez a simplified apply. No nem azért, mert könnyebb használni, hanem azért, mert a `lapply` mindig listát ad eredményül, és annak használata kicsit macerás. Azonban, ha vektoron futunk végig, vagy a listánk amúgy is könnyen vektorrá alakítható (mert mindegyik eleme ugyanolyan típusú), akkor a `lapply` eredményét nyugodtan leegyszerűsíthetjük vektorrá. A `sapply` nem csinál mást, mint futtat egy `lapply`-t, majd az eredményen egy `unlist`-et, és az eredményt visszadja.

```
unlist(test.list, use.names = F)
```

```
##  [1]   1   1   2   3  10  11  12  13  14  15  16  17  18  19  20  21  22
## [18]  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
## [35]  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56
## [52]  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73
## [69]  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
## [86]  91  92  93  94  95  96  97  98  99 100
```

Az unlist hívása egyszerűen egy nagy vektorrá alakította a listánkat, ami tartalmazza annak összes elemét.

Amennyiben az előző feladatot szeretnénk megoldani `sapply`-al:

```
sums <- sapply(test.list, FUN=sum)
sums
```

```
##    a    b    c
##    1    6 5005
```

Tehát végeredményképp vektort kapunk.

Létezik még igen sok egyéb apply függvény: vapply, mapply, rapply, tapply, replicate. Ezekeről egyrészt az R beépített súgója nyújthat bővebb információt, másrészt stackoverflow-n találhatunk egy kíváló magyarázatot

Vállalkozószelleműbbek utánanézhetnek a Map, Filter, Reduce függvényeknek is (ezek amúgy Java-ban, python-ban és JavaScriptben is megtalálhatók)

A legbátrabbak, akiknek már a fenti függvények ismerősek, betvehetik magukat a matematikai programozás mélyébe a `functional` csomggal.

Apropó csomagok...

Csomagok / könyvátrak

Mint láthattuk az R-be igen sok függvény be van építve, amiket mi magunk is meg tudnánk írni. Ez azért van, mert rengeteg problémával találkoztak a programozók, amióta programnyelvek léteznek, és ezekre rengeteg függvényt írtak, amiket sokszor beépítenek a programnyelvek alapkönnyvtáraiba – **standard library**-ébe.

Viszont vannak olyan problémák, amik ritkábban merülnek föl, nem mindenkinek jó, ha előre megír függvényekkel túlságosan felhízlalják az interpretert. Éppen ezért találták ki a csomagokat avagy könyvátárakat. Ezek olyan fájlok, ahol előre megírt függvényeket tárolnak, és közzé is teszik őket az interneten. Anyiban különzönek a saját `utils.R` szerű fájljainktól, hogy midneképpen dokumentáltak, minden függvényükhöz tartozik egy `?`-el hívható help, és beolvasásuk után nem jelennek meg a `Global Environment`ben.

`install.packages`

Telepíteni az `install.packages("packagename")` paranccsal tudjuk. Például az igen hasznos `dplyr` packageet az `install.packages("dplyr")` paranccsal tudjuk telepíteni. A `dplyr` egyébként nagyban megkönnyíti a nagymennyisgű táblázatos adatok átrendezését és szűrését.

`library` és `require`

Ha sikerült telepíteni a `library` és `require` parancsokkal tudjuk őket beolvasni. Mi lehetőleg használjuk `library` parancsot. Itt olvashattok többet arról, hogy miért